2 Quantum Computing Fundamentals

In this chapter, we describe the fundamental concepts and rules of quantum computing. In parallel, we develop an initial, easy-to-understand, and easy-to-debug code base for building and simulating smaller-scale algorithms.

The chapter is structured as follows. We first introduce our basic underlying data type, the Python `Tensor` type, which is derived from `numpy`'s `ndarray` data structure. Using this type we construct single qubits and quantum states composed of many qubits. We define operators, which allow us to modify states, and describe a range of important single-qubit gates. Controlled gates, which play a similar role to that of control flow in classical computing, come next. We detail how to describe quantum circuits via the Bloch sphere and in quantum circuit notation. A discussion of entanglement follows, that fascinating "spooky action at a distance," as Einstein called it. In quantum physics, measurement might be even more problematic than entanglement (Norsen, 2017). In this text, we avoid philosophy and conclude the chapter by describing a simple way to simulate measurements.

2.1 Tensors

Quantum mechanics and quantum computing are expressed in the language of linear algebra – vectors, matrices, and operations such as the dot product, outer product, and Kronecker products. As we develop the theory, we complement it with working code to allow experimentation.

We start by describing a fundamental data structure in Python. Python may be slow in general, but it also has the vectorized and accelerated `numpy` numerical library for scientific computing. We will make heavy use of this library so that we do not have to implement standard numerical linear algebra operations ourselves. In general, we follow Google's coding style guides for Python (Google, 2021b) and C++ (Google, 2021a).

The core data types, such as states, operators, and density matrices, are all vectors and matrices of complex numbers. It is good practice to base all of them on one common type abstraction, which hides the underlying implementation. This approach avoids potential problems with type mismatches and makes analysis, testing, debugging, pretty-printing, and other common functionality easier to maintain consistently. The base data type for all subsequent work will be a common `Tensor` class.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

We derive `Tensor` from the `ndarray` array data structure in `numpy`. Since we derive `Tensor`, it will behave just like a `numpy` array, but we can augment it with additional convenience functions.

There are several complex ways to instantiate an `ndarray`. The proper way to derive a class from this data type is complicated but well documented.[1] The implementation is in the open-source repository in `lib/tensor.py`:

```
import numpy as np

class Tensor(np.ndarray):
    """Tensor is a numpy array representing a state or operator."""

    def __new__(cls, input_array) -> Tensor:
        return np.asarray(input_array, dtype=tensor_type()).view(cls)

    def __array_finalize__(self, obj) -> None:
        if obj is None: return

        # np.ndarray has complex construction patterns. Because of this,
        # if new attributes are needed, add them like this:
        #    self.info = getattr(obj, 'info', None)
```

Note the use of `tensor_type()` in this code snippet: It abstracts the floating-point representation for complex numbers. Why do we do this? The choice of which complex data type to use is an interesting question, and each has its implications. Should it be complex numbers based on 64-bit doubles, 32-bit floats, or something else, for example, TPU's 16-bit bfloat format? Smaller data types are faster to simulate because of lower memory bandwidth requirements. But what level of accuracy is needed for which circuit? The `numpy` package supports `np.complex128` and `np.complex64`, so we simply define a global variable that holds this type's width. Having this information in one place makes it easy to experiment with different data types later on.

```
# Bit width of complex data type, 64 or 128.
tensor_width = 64

# All math in this package will use this base type.
# Valid values can be np.complex128 or np.complex64
def tensor_type():
  """Return complex type."""

  if tensor_width == 64:
    return np.complex64
  return np.complex128
```

[1] https://numpy.org/doc/stable/user/basics.subclassing.html.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

As we will see in our discussion of quantum states in Section 2.3, the Kronecker product of tensors, denoted with operator $\otimes$, is an important operation. This product is often referred to as *tensor product*. The *Kronecker product* describes a block product between matrices and is the correct term to use. We use the terms Kronecker product and tensor product interchangeably – *tensoring states* rolls off the tongue much more easily than *Kroneckering states*.

We implement it by adding the member function `kron` to the `Tensor` class. The function simply delegates to the function of the same name in `numpy`. We make heavy use of this operation, so for convenience we additionally overload the $\star$ operator to call this function.

There is the potential to confuse the $\star$ operator with simple matrix multiplication. However, in Python and with `numpy`, matrix multiplication is done with the *at* operator @. We inherit this operator from `numpy`; we don't have to implement it ourselves.

```
def kron(self, arg: Tensor) -> Tensor:
    """Return the Kronecker product of this object with arg."""

    return self.__class__(np.kron(self, arg))

 def __mul__(self, arg: Tensor) -> Tensor:
    """Inline * operator maps to Kronecker product."""

    return self.kron(arg)
```

In our initial approach to quantum computing, we will often construct larger matrices by tensoring together many identical matrices, which corresponds to calling the `kron` function multiple times. For example, to tensor together n unitary matrices U, we will use the following notation:

$$\underbrace{U \otimes U \otimes \cdots \otimes U}_{n} = U^{\otimes n}.$$

This is equivalent to a power function, but instead of multiplication we want Kronecker products. Naming is hard, but this function basically names itself. We should call it Kronecker power function – or `kpow` (pronounced "Ka-Pow"). We handle cases where the exponent is 0 as a special case, as $x^0 = 1.0$. As expected, `numpy` computes the correct Kronecker product of a matrix with a scalar:

```
def kpow(self, n: int) -> Tensor:
    """Return the tensor product with itself `n` times."""

    if n == 0:
        return 1.0
    t = self
    for _ in range(n - 1):
        t = np.kron(t, self)
    return self.__class__(t)  # Necessary to return a Tensor type
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Often, especially during testing, we want to compare a `Tensor` to a given value. We are working with complex numbers, which are based on the double or float data types. Direct comparison of values of these types is bad practice due to floating-point precision issues. Instead, for equality, we have to check that the difference between two numerical values is smaller than a given ϵ. Fortunately, `numpy` offers the function `allclose()`, which compares full tensors, so we do not have to iterate over dimensions and compare the real and imaginary parts. We use a tolerance of 10^{-6}, here and in almost all other places, and add this method to our `Tensor` type:[2]

```
def is_close(self, arg) -> bool:
    """Check that a 1D or 2D tensor is numerically close to arg."""

    return np.allclose(self, arg, atol=1e-6)
```

Python's `math` module has an `isclose` function. However, we decided to follow Google's coding style, which requires naming those types of function with an underscore after the `is_`. This is what we have chosen to do here to keep the overall code consistent with the style guide.

In Section 1.4, we learned about Hermitian and unitary matrices. The two helper functions below check for these conditions:

```
def is_hermitian(self) -> bool:
    """Check if this tensor is Hermitian - Udag = U."""

    if len(self.shape) != 2:
        return False
    if self.shape[0] != self.shape[1]:
        return False
    return self.is_close(np.conj(self.transpose()))

def is_unitary(self) -> bool:
    """Check if this tensor is unitary - Udag*U = I."""

    return Tensor(np.conj(self.transpose()) @ self).is_close(
                  Tensor(np.eye(self.shape[0])))
```

Some of the matrices we will encounter later in the text are *permutation matrices*, which have only a single 1 in each row and column of the matrix. This routine verifies this property:

[2] Note that for scalars, `math.isclose` is significantly faster than `np.allclose`. We will use it in performance-critical code.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
def is_permutation(self) -> bool:
    x = self
    return (x.ndim == 2 and x.shape[0] == x.shape[1] and
            (x.sum(axis=0) == 1).all() and
            (x.sum(axis=1) == 1).all() and
            ((x == 1) or (x == 0)).all())
```

2.2 Qubits

In classical computing, a bit can have the value 0 or 1. If we think of a bit as a switch, it is either off or on. You could say it is in the off-state (0-state) or on-state (1-state). Quantum bits, which are also called *qubits*, can be in a 0-state or a 1-state as well. What makes them quantum is that they are in a superposition of these states: They can be in the 0-state and the 1-state at the same time. What does this mean exactly?

First, we have to distinguish between a *qubit* and the *state of a qubit*. Physical qubits, developed for real quantum computers, are real physical entities, such as ions captured in an electric field, Josephson junctions on an ASIC, and so on. The state of a qubit describes some measurable property of that qubit, for example, the energy level of an electron. In quantum computing, at the level of programming abstractions, the physical implementation does not matter; instead, we are concerned with the measurable state. This is similar to classical computing, where very few people are knowledgeable about the quantum effects that enable transistors at the level of logic gates. Thus, the terms qubit and state of the qubit are used interchangeably; we typically only use the term qubit.

This state space of one or more qubits is often denoted by the Greek symbol $|\psi\rangle$ ("psi"). The standard notation for a qubit's 0-state is $|0\rangle$ in the Dirac notation. The 1-state is correspondingly written as $|1\rangle$. You can think of these as physically distinguishable states, such as the spin of an electron. These two states, $|0\rangle$ and $|1\rangle$, are known as *basis states*. We will *not* delve into the typical elaborate discussion of linear algebra and the theory of vector spaces here. All we need to know is that basis states represent orthogonal sets of vectors of dimensionality n (vectors with n components). *Any* vector of the same dimensionality can be constructed from linear combinations of basis states. In our context, we also require that basis vectors are normalized, forming an *orthonormal* set of basis vectors. Another way to say this is that the basis vectors are linearly independent and have a modulus of 1.

Superposition now simply means that the state of a qubit is a *linear combination* of orthonormal basis states, for example the $|0\rangle$ and $|1\rangle$ states:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where α and β are complex numbers, called the *probability amplitudes*, with $|\alpha|^2 + |\beta|^2 = 1$.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Note that we use the square of the norm, not the square of a complex number. As will become clear later, this follows from one of the fundamental postulates of quantum mechanics: On measurement, the state collapses to either $|0\rangle$ with (real) probability $|\alpha|^2$ or $|1\rangle$ with (real) probability $|\beta|^2$. The state has to collapse to one of the two, and thus, the added probabilities must add up to 1.0. If both α and β are exactly $\sqrt{1/2}$, there is an equal $\sqrt{1/2}^2 = 1/2$ probability that the state collapses to $|0\rangle$ or $|1\rangle$ on measurement. If α is 1.0 and β is 0.0, it is certain that the state will collapse to the $|0\rangle$ state on measurement.

Let us look at a standard example. Given a qubit $|\phi\rangle$ as

$$|\phi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{i}{2}|1\rangle,$$

the probability of measuring $|0\rangle$ is

$$Pr_{|0\rangle}(|\phi\rangle) = \left(\frac{\sqrt{3}}{2}\right)\left(\frac{\sqrt{3}}{2}\right) = \frac{3}{4}.$$

The probability of measuring $|1\rangle$ is the following – we compute the norm squared of the factor $i/2$:

$$Pr_{|1\rangle}(|\phi\rangle) = \left|\frac{i}{2}\right|^2 = \left(\frac{-i}{2}\right)\left(\frac{i}{2}\right) = \frac{1}{4}.$$

The following code will translate these concepts into a straightforward implementation. As a forward reference, we use the type `State`, which we discuss in the next section. Put simply, a `State` is a vector of complex numbers implemented using `Tensor`.

To construct a qubit, we need either α or β, or both. If only one is provided, we can easily compute the other one, given that their squared norms must add to 1.0. To compute the norms of the complex numbers α and β, we multiply each one with its complex conjugate (hence the use of `np.conj()`). The result will be a real number. For the code not to generate a type error from `numpy`, we have to explicitly convert the result with `np.real()`. We compare the results to 1.0, and if it is within tolerance, we construct and return the qubit.

What data structure should we use to represent a qubit? We simply create an array of two complex values, fill in α and β, and return a `State` constructed from this array.

```
def qubit(alpha: Optional[np.complexfloating] = None,
          beta: Optional[np.complexfloating] = None) -> State:
    """Produce a given state for a single qubit."""

    if alpha is None and beta is None:
        raise ValueError('alpha, or beta, or both are required')

    if beta is None:
        beta = math.sqrt(1.0 - np.conj(alpha) * alpha)
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
    if alpha is None:
        alpha = math.sqrt(1.0 - np.conj(beta) * beta)

    if not math.isclose(np.conj(alpha) * alpha +
                        np.conj(beta) * beta, 1.0):
        raise ValueError('Qubit probabilities do not add to 1.')

    qb = np.zeros(2, dtype=tensor.tensor_type())
    qb[0] = alpha
    qb[1] = beta
    return State(qb)
```

From this code, you can infer what the basis states might look like – a state is just a complex vector: the $|0\rangle$ state should be $[1,0]^T$ and the $|1\rangle$ state $[0,1]^T$. With this in mind, the state of a qubit can be written in these forms:

$$\begin{aligned}|\psi\rangle &= \alpha|0\rangle + \beta|1\rangle \\ &= \alpha\begin{bmatrix}1\\0\end{bmatrix} + \beta\begin{bmatrix}0\\1\end{bmatrix} \\ &= \begin{bmatrix}\alpha\\\beta\end{bmatrix}.\end{aligned}$$

The choice of $|0\rangle = [1,0]^T$ and $|1\rangle = [0,1]^T$ as the basis states is not the only possible one. What matters is that these vectors are orthonormal. They are orthogonal, with a mutual scalar product of $\langle 0|1\rangle = 0.0$, and normalized, with scalar products of $\langle 0|0\rangle = 1.0$ and $\langle 1|1\rangle = 1.0$.

The set of orthonormal bases $[1,0]^T$ and $[0,1]^T$ for the qubit vector space, which is also called the *computational basis*, is intuitive and simplifies the math. But other bases are possible, especially the ones resulting from rotations. Those are commonplace in quantum computing, as we will see shortly.

2.3 States

As we saw in the previous section, qubits are *states*, vectors of complex numbers representing probability amplitudes. We should use our trusty `Tensor` class to represent states in code. We inherit a `State` class from `Tensor` and add a moderately improved print function. The sources are in the open-source repository in `lib/state.py`:

```
class State(tensor.Tensor):
    """class State represents single and multi-qubit states."""

    def __repr__(self) -> str:
        s = 'State('
        s += super().__str__().replace('\n', '\n' + ' ' * len(s))
        s += ')'
        return s
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
    def __str__(self) -> str:
        s = f'{self.nbits}-qubit state.'
        s += ' Tensor:\n'
        s += super().__str__()
        return s
```

The state of two or more qubits is defined as their tensor product. To compute it, we added the `*` operator to the underlying `Tensor` type in the previous section (implemented as the corresponding Python `__mul__` member function). Note that the tensor product of two states, which both have a norm of 1.0, also has a norm of 1.0.

For two qubits $|\phi\rangle$ and $|\chi\rangle$, the combined state can be written as in Equation (1.3), with $\otimes$ being the symbol for the Kronecker product:

$$|\psi\rangle = |\phi\rangle \otimes |\chi\rangle = |\phi\rangle|\chi\rangle = |\phi, \chi\rangle = |\phi\chi\rangle.$$

Given this definition, the state for n qubits is a `Tensor` of 2^n complex numbers, the probability amplitudes. We could maintain this length as an extra member variable to `State`, but it is easy to compute from the length of the state vector (which is already maintained by `numpy`). We define it as a property. Because this property is required for all classes derived from `Tensor` (e.g., `States` and `Operators`), we add the `nbits` property to the `Tensor` base class, so that derived classes can inherit it:

```
@property
def nbits(self) -> int:
    """Compute the number of qubits in the state."""

    return int(math.log2(self.shape[0]))
```

Python does have a `bit_length()` function to determine how many bits are needed to represent a number. Here, using this function would be wrong. To represent eight states, $n = 3$ for a state of 2^n complex numbers. However, you would need four classical binary bits to represent the number 8. Using a value of $n - 1$ will not work for an input value of 0. Additionally, `bit_length()` returns values for negative numbers, which makes no sense in this quantum state context. For all these reasons, we decided to use the `log2` function. As a code example, let us combine two qubits:

```
   psi = state.qubit(alpha=1.0)    # corresponds to |0>
   phi = state.qubit(beta=1.0)     # corresponds to |1>
   combo = psi * phi
   print(combo)
>>
2-qubit state. Tensor:
[0.+0.j 1.+0.j 0.+0.j 0.+0.j]
```

The resulting state is a `Tensor` with the complex values $[0.0, 1.0, 0.0, 0.0]$, which is the result you would expect from tensoring $[1, 0]^T$ and $[0, 1]^T$. The index of the value

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

1.0 is 1, which indicates that the combination of the $|0\rangle$ state and the $|1\rangle$ state is being interpreted as binary `0b01`. This will become more clear in the next section, where we discuss qubit ordering.

In all code examples that follow, states are constructed from the high order bit to the low order bit. This choice is arbitrary, and as a matter of fact, some texts have it the other way around. The important thing is to stay consistent.

A quantum state's probability amplitudes represent probabilities – the squared norms of all amplitudes must add up to 1.0. Basis states are *normalized*. As an example (which generalizes to n qubits), for two qubits there are four basis states, and we can write the state $|\psi\rangle$ as a superposition:

$$
\begin{aligned}
|\psi\rangle &= c_0 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + c_1 \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + c_2 \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + c_3 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\
&= c_0 |\psi_0\rangle + c_1 |\psi_1\rangle + c_2 |\psi_2\rangle + c_3 |\psi_3\rangle = \sum_{i=0}^{3} c_i |\psi_i\rangle.
\end{aligned}
$$

The amplitudes are complex numbers, so to compute the norm we multiply by the complex conjugates:

$$
\begin{aligned}
\langle\psi|\psi\rangle &= c_0^* \langle\psi_0| c_0 |\psi_0\rangle + c_1^* \langle\psi_1| c_1 |\psi_1\rangle + \cdots + c_n^* \langle\psi_{n-1}| c_n |\psi_{n-1}\rangle \\
&= c_0^* c_0 \langle\psi_0| \psi_0\rangle + c_1^* c_1 \langle\psi_1| \psi_1\rangle + \cdots + c_n^* c_n \langle\psi_{n-1}| \psi_{n-1}\rangle \\
&= c_0^* c_0 + c_1^* c_1 + \cdots + c_{n-1}^* c_{n-1} \\
&= 1.0.
\end{aligned}
$$

For states that are products of states, we apply Equation (1.7). Note, again, that the elements in the bras are the complex conjugates:

$$
\begin{aligned}
|\psi\rangle &= |\phi\chi\rangle \\
\langle\psi| &= |\phi\chi\rangle^\dagger = \langle\phi\chi| \\
\langle\psi|\psi\rangle &= \langle\phi\chi|\phi\chi\rangle = \langle\phi|\phi\rangle\langle\chi|\chi\rangle.
\end{aligned}
$$

You can write a test like this to confirm:

```
p1 = state.qubit(alpha=random.random())
x1 = state.qubit(alpha=random.random())
psi = p1 * x1  # Tensor product.

# inner product of full state
self.assertTrue(np.allclose(np.inner(psi.conj(), psi), 1.0))

# inner product of the constituents multiplied
self.assertTrue(np.allclose(np.inner(p1.conj(), p1) *
                np.inner(x1.conj(), x1), 1.0))
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

2.3.1 Qubit Ordering

The ordering of qubits during construction, access to results, and conversion of binary strings are important and can be sources of problems if not understood properly. For this text, the key points to internalize are the following:

- As qubits are added to a circuit, they are being added from left to right (in a binary string), from the high-order qubit to the low-order qubit.
- In Dirac notation, two tensored states are written as $|x, y\rangle$, for example, $|0, 1\rangle$. Additionally, in this notation the most significant bit is the first to appear. States like this are also expressed as decimals. Here, the state is interpreted as $|1\rangle$ (binary `01`) as opposed to $|2\rangle$ (binary `10`) if incorrectly read from right to left:

$$\underbrace{|0\rangle}_{\text{High-order}} \otimes \cdots \otimes \underbrace{|0\rangle}_{\text{Low-order}}$$

- We will see in Section 2.8 that circuits are drawn as a vertical stack of qubits, with the top qubit being the most significant.
- We will learn soon about simple functions for constructing composite states from $|0\rangle$ and $|1\rangle$ states. In these functions, the first qubit to appear will be the most significant qubit, similar to the circuit notation. For example, the state $|\psi\rangle = |1\rangle \otimes |0\rangle \otimes 1 \otimes |0\rangle$ is constructed with

```
psi = state.bitstring(1, 0, 1, 0)
```

- When states are formatted to print, the most significant bit will also be to the left, as in binary notation.
- We have to distinguish between how bits or qubits are interpreted and how we store them in our programs. In classical computing, bit 0 is typically the rightmost bit, which is the least significant bit. When we *store* bits as an array of bits, we will store them from low index to high array index. This means the index into the array is 0 for the first stored bit. In the quantum case, this is the most significant qubit.

  This is a constant source of confusion, not just in this context but in any context that has to represent bits or qubits in an indexed, array-like data structure.

2.3.2 Binary Interpretation

We can write tensor products of $|0\rangle$ and $|1\rangle$ states as in this example:

$$|0\rangle \otimes |1\rangle \otimes |1\rangle = |011\rangle.$$

For brevity, when interpreting the bitstrings as binary numbers and numbering them in decimal, states are often written as in this example:

$$|011\rangle = |3\rangle.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Be aware of the potential for confusion between the state $|000\rangle$, the corresponding decimal state $|0\rangle$, and an actual one-qubit state $|0\rangle$. How does the decimal interpretation of a state relate to the state vector?

- The state $|00\rangle$ is computed as $\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$. Also called $|0\rangle$.
- The state $|01\rangle$ is computed as $\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$. Also called $|1\rangle$.
- The state $|10\rangle$ is computed as $\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$. Also called $|2\rangle$.
- The state $|11\rangle$ is computed as $\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$. Also called $|3\rangle$.

To find the probability amplitude for a given state, we can use binary addressing. The state vector for the three-qubit state $|011\rangle$ is:

$$[0,0,0,1,0,0,0,0]^T.$$

Interpreting the right-most qubit in $|011\rangle$ as the least significant bit 0 with a bit value of 1, the middle one as bit 1 with a bit value of 1 and a power-of-2 value of 2, and the left-most as bit 2 with a value of 0, the state $|011\rangle$ computes the value 3 in decimal, state $|3\rangle$. We index the state vector as an array, as described above, from left to right, from 0 to n. Indeed, entry 3 of the state vector is set to 1. The amplitude for each state in the state vector can be found with this simple binary addressing scheme.

Note that the tensor product representation of this 3-qubit state contains the amplitudes for all eight possible states. Seven states have an amplitude of 0.0. This already hints at a potentially more efficient sparse representation.

2.3.3 Member Functions

We now introduce a few important member functions of the `State` type. This code uses functions from the `helper` module, which is described in Section 2.4.

Now that we understand the order of qubits and state vector indexing, we can add functions to `State` to return the amplitude and probability of a given state. The probability is a real number, but we still have to cast it to an actual real number via `np.real` to avoid warning messages on type conflicts.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
def ampl(self, *bits) -> np.complexfloating:
    """Return amplitude for state indexed by 'bits'."""

    idx = helper.bits2val(bits)  # in helper.py
    return self[idx]

def prob(self, *bits) -> float:
    """Return probability for state indexed by 'bits'."""

    amplitude = self.ampl(*bits)
    return np.real(amplitude.conj() * amplitude)
```

We use Python parameters that are decorated with a $*$. This means a variable number of arguments is allowed and the parameters are *packed* as a tuple. To unpack the tuple, you have to prefix the access with a $*$ again, as shown in the function definitions above.

As an example, for a four-qubit state, you can get the amplitude and probability for the state $|1011\rangle$ in the following way:

```
psi.ampl(1, 0, 1, 1)
psi.prob(1, 0, 1, 1)
```

The following snippet iterates over all possible states and prints the probabilities for each state:

```
    for bits in helper.bitprod(4):
        print(psi.prob(*bits))
```

During algorithm development, we often want to find the one state with the highest probability. For this, we add the following convenience function, which iterates over all possible states and returns the state/probability pair with the highest probability:

```
def maxprob(self) -> (List[float], float):
    """Find state with highest probability."""

    maxbits, maxprob = [], 0.0
    for bits in helper.bitprod(self.nbits):
        cur_prob = self.prob(*bits)
        if cur_prob > maxprob:
            maxbits, maxprob = bits, cur_prob
    return maxbits, maxprob
```

As we will see later, it can become necessary to renormalize a state vector. This is done with the `normalize` member function. Note that this function assumes that the dot product is not 0.0; otherwise, this code will result in a division by zero exception:

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
def normalize(self) -> None:
    """Renormalize the state. Sum of norms==1.0."""

    dprod = np.conj(self) @ self
    if (dprod.is_close(0.0)):
        raise AssertionError('Normalizing to zero-probability state.')
    self /= np.sqrt(np.real(dprod))
```

The *phase* of a qubit is the angle obtained when converting the qubit's complex amplitude to polar coordinates. We only use this during print-outs, so we convert the phase to degrees here.

```
def phase(self, *bits) -> float:
    """Compute phase of a state from the complex amplitude."""

    amplitude = self.ampl(*bits)
    return math.degrees(cmath.phase(amplitude))
```

Finally, to help in debugging, it is always helpful to have a dumper function that lists all of a state's relevant information. By default, this function only prints the states with nonzero probability. (Set parameter `prob_only` to `False` to see all states). An optional description string can be passed in as well.

```
def state_to_string(bits) -> str:
  """Convert state to string like |010>."""

  s = ''.join(str(i) for i in bits)
  return '|{:s}> (|{:d}>)'.format(s, int(s, 2))

def dump_state(psi, desc: Optional[str]=None,
               prob_only: bool=True) -> None:
  """Dump probabilities for a state, as well as local qubit state."""

  if desc:
    print('|', end='')
    for i in range(psi.nbits-1, -1, -1):
      print(i % 10, end='')
    print(f'> \'{desc}\'')

  state_list: List[str] = []
  for bits in helper.bitprod(psi.nbits):
    if prob_only and (psi.prob(*bits) < 10e-6):
      continue
    state_list.append(
        '{:s}:  ampl: {:+.2f} prob: {:.2f} Phase: {:5.1f}'
        .format(state_to_string(bits),
                psi.ampl(*bits),
                psi.prob(*bits),
                psi.phase(*bits)))
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
  state_list.sort()
  print(*state_list, sep='\n')

def dump(self, desc: Optional[str] = None,
         prob_only: bool = True) -> None:
    dump_state(self, desc, prob_only)
```

As an example, the output from the dumper may look like the following, showing all states with nonzero probability:

```
|001> (|1>):  ampl: +0.50+0.00j prob: 0.25 Phase:   0.0
|011> (|3>):  ampl: +0.35+0.35j prob: 0.25 Phase:  45.0
|101> (|5>):  ampl: +0.00+0.50j prob: 0.25 Phase:  90.0
|111> (|7>):  ampl: -0.35+0.35j prob: 0.25 Phase: 135.0
```

2.3.4 State Constructors

Using the methods described so far, let us define standard constructors to create composite states. The first two functions are for states consisting of all $|0\rangle$ and $|1\rangle$.

```
# The functions zeros() and ones() produce the all-zero or all-one
# computational basis vector for `d` qubits, i.e.,
#     |000...0> or
#     |111...1>
#
# The result of this tensor product is
#   always [1, 0, 0, ..., 0]^T or [0, 0, 0, ..., 1]^T
#
def zeros_or_ones(d: int = 1, idx: int = 0) -> State:
    """Produce the all-0/1 basis vector for `d` qubits."""

    if d < 1:
        raise ValueError('Rank must be at least 1.')
    shape = 2**d
    t = np.zeros(shape, dtype=tensor.tensor_type())
    t[idx] = 1
    return State(t)

def zeros(d: int = 1) -> State:
    """Produce state with 'd' |0>, eg., |0000>."""
    return zeros_or_ones(d, 0)

def ones(d: int = 1) -> State:
    """Produce state with 'd' |1>, eg., |1111>."""
    return zeros_or_ones(d, 2**d - 1)
```

The function `bitstring` allows the construction of states from a defined series of $|0\rangle$ and $|1\rangle$ states. As noted above, the most significant bit comes first:

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
def bitstring(*bits) -> State:
    """Produce a state from a given bit sequence, eg., |0101>."""

    d = len(bits)
    if d == 0:
        raise ValueError('Rank must be at least 1.')
    t = np.zeros(1 << d, dtype=tensor.tensor_type())
    t[helper.bits2val(bits)] = 1
    return State(t)
```

Sometimes, in particular for testing or benchmarking, we want to generate a random combination of n $|0\rangle$ and $|1\rangle$ states:

```
def rand(n: int) -> State:
    """Produce random combination of |0> and |1>."""

    bits = [0] * n
    for i in range(n):
        bits[i] = random.randint(0, 1)
    return bitstring(*bits)
```

Finally, because the canonical single-qubit states $|0\rangle$ and $|1\rangle$ are used often, it may make sense to define constants for them. Global variables are bad style. We only added them to offer compatibility with other frameworks. Don't use them.

```
# These two are used so commonly, make them constants.
zero = zeros(1)
one = ones(1)
```

Can we initialize a state with a given normalized vector? Yes, we can. This is a pattern we will see later, in the section on phase estimation (Section 6.4), where we initialize a state directly with the eigenvector of a unitary matrix:

```
umat = scipy.stats.unitary_group.rvs(2**nbits)
eigvals, eigvecs = np.linalg.eig(umat)
psi = state.State(eigvecs[:, 0])
```

2.3.5 Density Matrix

For a given state $|\psi\rangle$, we can construct a *density matrix* by computing the outer product of a state with itself. For convenience, we add the function `density()` to our `State` class to compute this outer product. Typically, the Greek letter ρ ("rho") is used to denote a density matrix:

$$\rho = |\psi\rangle\langle\psi|.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

In code:

```
def density(self) -> tensor.Tensor:
    return tensor.Tensor(np.outer(self, self.conj()))
```

The theory of quantum computing can be expressed in terms of density matrices. Some important concepts can *only* be expressed with the help of these matrices. We will not spend much time on this part of the theory. We mention density matrices here because they make an appearance in Section 2.15 on measurement.

One property of density matrices is important in our context. Given that these matrices are being constructed from an outer product, it means that the diagonal elements contain the probabilities of measuring one of the basis states for $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$.

$$|\psi\rangle\langle\psi| = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}\begin{bmatrix}\alpha^* & \beta^*\end{bmatrix} = \begin{bmatrix}\boxed{\alpha\alpha^*} & \alpha\beta^*\\ \beta\alpha^* & \boxed{\beta\beta^*}\end{bmatrix}.$$

Because of the way we constructed this density matrix, it represents a pure state; it is not entangled or statistically mixed with anything else. Correspondingly, the trace of the density matrix is 1; it is the sum of all state probabilities.

2.4 Helper Functions

We will need a small set of helper functions. They will be used in several places, but don't seem to belong to any specific core module. Hence we collect helper functions in the open-source repository in file `lib/helper.py`.

Bit Conversions

We often have to convert between a decimal number and its binary representation as a tuple of 0s and 1s. These two helper functions make that easy:

```
def bits2val(bits: List[int]) -> int:
    """For a given enumerable 'bits', compute the decimal integer."""

    # We assume bits are given in high to low order. For example,
    # the bits [1, 1, 0] will produce the value 6.
    return sum(v * (1 << (len(bits)-i-1)) for i, v in enumerate(bits))

def val2bits(val: int, nbits: int) -> List[int]:
    """Convert decimal integer to list of {0, 1}."""

    # We return the bits in order high to low. For example,
    # the value 6 is being returned as [1, 1, 0].
    return [int(c) for c in format(val, '0{}b'.format(nbits))]
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Iteration over Bits

There will also be times when we want to iterate over all possible combinations of 0's and 1's, which is to enumerate all binary values of length `nbits`. Note the use of Python's `yield` construct below, which allows usage of this function in Python `for` loops.

```
def bitprod(nbits: int) -> Iterable[int]:
    """Produce the iterable cartesian of nbits {0, 1}."""

    for bits in itertools.product([0, 1], repeat=nbits):
        yield bits
```

2.5 Operators

We have discussed qubits and states. In quantum computing, how are these states *modified*? Classical bits are manipulated via logic gates, such as AND, OR, XOR, and NAND. In quantum computing, qubits and states are changed with *operators*. It seems appropriate to think of operators as the Instruction Set Architecture (ISA) of a quantum computer. It is a different ISA than that of a typical classical computer, but it is an ISA nonetheless. It enables computation.

In this section, we discuss operators, their structure, properties, and how to apply them to states. All sources are in the open-source repository, in file `lib/ops.py`.

2.5.1 Unitary Operators

Any unitary matrix of dimension 2^n can be considered a quantum operator. Operators are also called *gates*, in analogy to classical logic gates. Unitary matrices applied to state vectors do not change the modulus of the state vector; they are *norm preserving*. A state vector represents probabilities as probability amplitudes. Applying an operator to this state might change individual states' amplitudes but must not change the fact that all probabilities must still add up to 1.0. This is important enough to warrant a proof.

Proof To show that U is norm-preserving, we need to show that $\langle Uv|Uw\rangle = \langle v|w\rangle$. This is to show that if U preserves the inner product structure, it must also preserve the norm:

$$\langle Uv|Uw\rangle = (v^\dagger U^\dagger)(Uw) = v^\dagger(U^\dagger U)w.$$

Now, $v^\dagger(U^\dagger U)w = v^\dagger w = \langle v, w\rangle$ implies that $(U^\dagger U) = I$. Any operator that preserves the norm must be unitary. □

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

An example of a single-qubit gate is the Identity gate, which, when applied, leaves a qubit unmodified:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}.$$

Another example is the X-gate (a synonym for the Pauli X-gate described in Section 2.6.2), which swaps the probability amplitudes of a qubit:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}.$$

We detail many standard gates later in this section. Note that because $UU^\dagger = I$, unitary matrices are necessarily invertible. As a result, all (unitary) quantum gates are reversible by simply using a gate's conjugate transpose.

On the other hand, Hermitian matrices are not necessarily unitary. In Section 2.15 we will see how Hermitian operators used for measurements are neither unitary nor reversible.

2.5.2 Base Class

Since operators are matrices, we derive them from our `Tensor` base class and provide the usual print functions:

```
class Operator(tensor.Tensor):
  """Operators are represented by square, unitary matrices."""

  def __repr__(self) -> str:
    s = 'Operator('
    s += super().__str__().replace('\n', '\n' + ' ' * len(s))
    s += ')'
    return s

  def __str__(self) -> str:
    s = f'Operator for {self.nbits}-qubit state space.'
    s += ' Tensor:\n'
    s += super().__str__()
    return s
```

To compute the adjoint with help of `numpy` is painless:

```
def adjoint(self) -> Operator:
    return Operator(np.conj(self.transpose()))
```

The `numpy` package has routines to print arrays, but we add another dumper function that produces a more compact output, making it easier to see the matrix structure instead of seeing values with high precision. This function can be adapted quickly to help during challenging debugging sessions.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
  def dump(self,
           description: Optional[str] = None,
           zeros: bool = False) -> None:
    res = ''
    if description:
      res += f'{description} ({self.nbits}-qubits operator)\n'
    for row in range(self.shape[0]):
      for col in range(self.shape[1]):
        val = self[row, col]
        res += f'{val.real:+.1f}{val.imag:+.1f}j  '
      res += '\n'
    if not zeros:
      res = res.replace('+0.0j', '    ')
      res = res.replace('+0.0', ' -  ')
      res = res.replace('-0.0', ' -  ')
      res = res.replace('+', ' ')
    print(res)
```

Here are examples for a two-qubit operator, printed both with this dumper function and `numpy`'s own print[3] function.

```
# dump
 0.5       0.5       0.5       0.5
 0.5      -0.5       0.5      -0.5
 0.5       0.5      -0.5      -0.5
 0.5      -0.5      -0.5       0.5

# numpy print
Operator for 2-qubit state space. Tensor:
[[ 0.49999997+0.j  0.49999997+0.j  0.49999997+0.j  0.49999997+0.j]
 [ 0.49999997+0.j -0.49999997+0.j  0.49999997+0.j -0.49999997+0.j]
 [ 0.49999997+0.j  0.49999997+0.j -0.49999997+0.j -0.49999997+0.j]
 [ 0.49999997+0.j -0.49999997+0.j -0.49999997+0.j  0.49999997-0.j]]
```

2.5.3 Operator Application

The application of a unitary operator to a state vector is a matrix-vector multiplication (operators are matrices, states are vectors). In Python, we define the function call operator `()` for this purpose. If we have a gate `X` and a state `psi`, we can apply `X` with `new_psi = ops.X(psi)`. The `__call__` function itself just wraps the `apply` function, which we define next. Note that this function accepts a state as well as an operator as its argument.

[3] `numpy` has a fairly flexible way to configure prints as well; see https://numpy.org/doc/stable/reference/generated/numpy.set_printoptions.html.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
def __call__(self,
             arg: Union[state.State, ops.Operator],
              idx: int = 0) -> state.State:
    return self.apply(arg, idx)
```

In the following, we gradually build up the function to apply an operator to a state. The initial version is fairly incomplete. We apply an operator to a state vector by using `numpy`'s matrix multiplication function `np.matmul`:

```
def apply(self,
          arg: Union[state.State, ops.Operator],
          idx: int) -> state.State:
    """Apply operator to a state or operator."""

    [...]
    if not isinstance(arg, state.State):
      raise AssertionError('Invalid parameter, expected State.')
    [...]
    return state.State(np.matmul(self, arg))
```

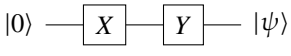
We can also apply an operator to another operator. In this case, application results in matrix-matrix multiplication. What is the order of applications when multiple operators are being applied in sequence?

Assume we have an X-gate and a Y-gate (to be explained later), and we want to apply them in sequence. We can write this the following way in Python, where gates are applied to a state, and we return the updated state:

```
    psi_1 = X(psi_0)
    psi_2 = Y(psi_1)
```

These are assignments, not to be confused with a mathematical notation like $x = y$, which expresses an equivalence. In Python, we could omit the indices and overwrite a single state variable `psi`.

In quantum circuit notation, which we explain in more detail below, this looks like the following, with time flowing from left to right:



In the function call notation, we would write the symbols from left to right as well. But note that function parameters are being evaluated first, which means they are being applied first:

```
      # The function call should mirror this semantic
      #   X(Y)
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

This already points to the fact that if we express the combined operator as the product of matrices, we have to invert their order (with @ being the matrix multiply operator in Python):

```
      # But in a combined operator matrix, Y comes first:
      #   (Y @ X)(psi)
```

This leads to the following, still incomplete implementation of `apply`, assuming the sizes of operator and state vector match:

```
def apply(self,
          arg: Union[state.State, ops.Operator],
          idx: int) -> Union[state.State, ops.Operator]:
    """Apply operator to a state or operator."""

    if isinstance(arg, Operator):
      if self.nbits != arg.nbits:
        raise AssertionError('Operator with mis-matched dimensions.')

      # Note: We reverse the order in this matmul. So:
      #   X(Y) == Y @ X
      #
      # This is to mirror that for a circuit like this:
      #   --- X --- Y --- psi
      #
      # Incrementally updating states we would write:
      #   psi = X(psi)
      #   psi = Y(psi)
      #
      # But in a combined operator matrix, Y comes first:
      #   (YX)(psi)
      #
      # The function call should mirror this semantic, since parameters
      # are typically evaluated first (and this mirrors the left to
      # right in the circuit notation):
      #   X(Y) = YX
      #
      return arg @ self

    if not isinstance(arg, state.State):
      raise AssertionError('Invalid parameter, expected State.')

    # Note the reversed order compared to above.
    return state.State(np.matmul(self, arg))
```

2.5.4 Multiple Qubits

The code above makes it possible to apply a gate to a single qubit. How does this work if we have a state of three qubits, which is a state of 2^3 complex numbers, and we want

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

to apply the 2×2 gate to just one qubit in their tensor product? The key property of the tensor product that enables handling this case is Equation (1.6):

$$(A \otimes B)(\alpha \otimes \beta) = (A \otimes \alpha)(B \otimes \beta).$$

We can utilize this equation with the identity gate I, which is the matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Applying I to any qubit leaves the qubit intact. The above equation allows us to, for example, apply the X-gate (discussed earlier) to the second qubit in a three-qubit state by tensoring together I, the X-gate, and another I to obtain an 8×8 matrix:

```
psi = state.bitstring(0, 0, 0)
op = ops.Identity() * ops.PauliX() * ops.Identity()
psi = op(psi)
psi.dump()
```

In Dirac notation, this is the original state:

$$|\psi_0\rangle = |0\rangle \otimes |0\rangle \otimes |0\rangle.$$

Interpreting $|000\rangle$ as the binary number 0 (remember, the least significant bit is to the right), this means that element 0 of the state vector of 8 elements should be a 1.0, which we can confirm by dumping the state:

```
1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

To apply the X-gate to qubit 1 in this way:

$$|\psi_1\rangle = (I \otimes X \otimes I)\,|\psi_0\rangle.$$

This becomes, according to Equation (1.6):

$$|\psi_1\rangle = I|0\rangle \otimes X|0\rangle \otimes I|0\rangle.$$

The X-gate flips the probability amplitudes. Another way to say this colloquially is that it flips a state from $|0\rangle$ to $|1\rangle$ (or from $|1\rangle$ to $|0\rangle$). So, applying the gates above will result in the modified state $|\psi_1\rangle$:

$$|\psi_1\rangle = |0\rangle \otimes |1\rangle \otimes |0\rangle.$$

This means, interpreting $|010\rangle$ as the binary number 2, that the state vector element 2 should now have the value 1.0, and indeed it does:

```
0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

To apply multiple operators in sequence, their individual expanded operators can be multiplied together to build a single, combined operator. For example, to apply the X-gate to qubit 1 and the Y-gate to qubit 2, we write:

```
psi = state.bitstring(0, 0, 0)
opx = ops.Identity() * ops.PauliX() * ops.Identity()
opy = ops.Identity() * ops.Identity() * ops.PauliY()
big_op = opx(opy)
psi = big_op(psi)
```

Note that there is a shortcut notation for this. To indicate that gate A should be applied to a qubit at a certain index i, we just write A_i. This notation means that this operator is being padded from both sides with identity matrices. For the example above, to apply the X-gate to qubit 1 and the Y-gate to qubit 2, we would write $X_1 Y_2$.

Of course, in regards to performance, constructing the full, combined operator up front for n qubits is the worst possible case, as we have to perform full matrix multiplies with matrices of size $(2^n)^2$. Matrix multiplication is of cubic[4] complexity $O(n^3)$. Since a matrix-vector product is of complexity $O(n^2)$, it can be faster to apply the gates individually, depending on the number of gates.

```
psi = state.bitstring(0, 0, 0)
opx = ops.Identity() * ops.PauliX() * ops.Identity()
psi = opx(psi)
opy = ops.Identity() * ops.Identity() * ops.PauliY()
psi = opy(psi)
```

Of course, in this particular example we could have simply combined the gates:

```
psi = state.bitstring(0, 0, 0)
opxy = ops.Identity() * ops.PauliX() * ops.PauliY()
psi = opxy(psi)
```

2.5.5 Operator Padding

Having to "pad" operators with identity matrices on the left and right is annoying and error-prone. Wouldn't it be more convenient to just apply an operator to a qubit at index `idx` and have the infrastructure do the rest for us? This is what *operator padding* does, which we will implement next.

[4] This is an approximation to make a point, which we will use in several places. More efficient algorithms are known, such as the Coppersmith–Winograd algorithm with complexity $O(2^{2.3752477})$.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

To apply a given gate, say the X-gate, to a state `psi` at a given qubit index `idx`, we write:

```
X = ops.PauliX()
psi = X(psi, idx)
```

To achieve this, we augment the function call operator for `Operator`. If an index is provided as a parameter, we pad the operator up to this index with identity matrices. Then, we compute the size of the given operator, which can be larger than 2×2, and if the resulting matrix's dimension is still smaller than the state it is applied to, we pad it further with identity matrices. In above example, instead of:

```
psi = state.bitstring(0, 0, 0)
opx = ops.Identity() * ops.PauliX() * ops.Identity()
psi = opx(psi)
```

we can now write the following. Note that the first pair of parentheses to `PauliX()` returns a simple 2×2 `Operator` object. The parenthesis `(psi, 1)` are the parameters passed to the operator's function call operator `__call__`, which delegates to the `apply` function. This is where the automatic padding finally happens. This syntax may be confusing on first sight:

```
psi = state.bitstring(0, 0, 0)
psi = ops.PauliX()(psi, 1)
```

With this, we can finalize the implementation of `apply`:

```
def apply(self,
          arg: Union[state.State, ops.Operator],
          idx: int) -> Union[state.State, ops.Operator]:
    """Apply operator to a state or operator."""

    if isinstance(arg, Operator):
      arg_bits = arg.nbits
      if idx > 0:
        arg = Identity().kpow(idx) * arg
      if self.nbits > arg.nbits:
        arg = arg * Identity().kpow(self.nbits - idx - arg_bits)

      if self.nbits != arg.nbits:
        raise AssertionError('Operator(O) with mis-matched dimensions.')

      #
      # [... Comment block as shown above]
      #
      return arg @ self
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
    if not isinstance(arg, state.State):
      raise AssertionError('Invalid parameter, expected State.')

    op = self
    if idx > 0:
      op = Identity().kpow(idx)  * op
    if arg.nbits - idx - self.nbits > 0:
      op = op * Identity().kpow(arg.nbits - idx - self.nbits)

    return state.State(np.matmul(op, arg))
```

2.6 Single-Qubit Gates

In this section, we list single-qubit gates that are commonly used in quantum computing. They are equivalent to logic gates in classical computing – understanding the basic gates helps in building up more sophisticated circuits. Quantum gates are similar. You have to understand their function in order to compose them into more interesting circuits. However, for the most part, the gates' functions are quite different from classical gates.

We start with simple gates and then discuss the more complicated roots and rotations before discussing the important Hadamard gate, which puts qubits in a superposition of basis states.

For each gate, we define a constructor function and allow passing a *dimension parameter* `d`, which allows the construction of multi-qubit operators from the same underlying single-qubit gates. For example, for the identity gates in the previous example, instead of having to write

```
y2 = ops.Identity() * ops.Identity() * ops.PauliY()
```

we can write the more compact

```
y2 = ops.Identity(2) * ops.PauliY()
```

that computes the following tensor product. Note the subscript in Y_2, which indicates that the Y-gate should only be applied to qubit 2:

$$Y_2 = I \otimes I \otimes Y = I^{\otimes 2} \otimes Y.$$

2.6.1 Identity Gate

We have seen the identity gate before; it is this matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Applying this gate to a state leaves the state intact:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}.$$

In code it looks like this, a common pattern we use for almost all of the gate constructor functions:

```
def Identity(d: int = 1) -> Operator:
    return Operator(np.array([[1.0, 0.0], [0.0, 1.0]])).kpow(d)
```

2.6.2 Pauli Matrices

The three Pauli matrices are essential in quantum computing and have many uses, some of which we will discover as we go along. Pauli matrices are usually denoted as $\sigma_x, \sigma_y, \sigma_z$ or $\sigma_1, \sigma_2, \sigma_3$. Sometimes we want to add the identity matrix I as a first Pauli matrix σ_0.

The Pauli X-gate is also known as the X-gate, the quantum Not-gate, or just X for short. It is called a Not-gate because it seemingly "flips" basis states in the following way:

$$X|0\rangle = |1\rangle \quad \text{and} \quad X|1\rangle = |0\rangle.$$

This can look confusing, especially for beginners. To clarify, the basis states themselves remain unmodified; they represent physical states. The X-gate only swaps their probability amplitudes:

$$X|\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}.$$

So it changes $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ to $|\psi\rangle = \beta|0\rangle + \alpha|1\rangle$. This works for all values of α and β, including the cases where either α or β is 0 and the other is 1. In code:

```
def PauliX(d: int = 1) -> Operator:
    return Operator(np.array([[0.0j, 1.0], [1.0, 0.0j]])).kpow(d)
```

The Pauli Y-gate, or just Y-gate, looks like this:

$$Y|\psi\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} -i\beta \\ i\alpha \end{bmatrix}.$$

```
def PauliY(d: int = 1) -> Operator:
    return Operator(np.array([[0.0, -1.0j], [1.0j, 0.0]])).kpow(d)
```

The Pauli Z-gate, or just Z-gate, is also known as the phase-flip gate, as it inverts the sign of the second qubit component.

$$Z|\psi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ -\beta \end{bmatrix}.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

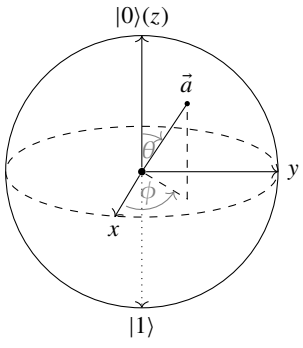


Figure 2.1 A Bloch sphere with axes x, y, and z.

In other words, it changes $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ to $|\psi\rangle = \alpha|0\rangle - \beta|1\rangle$. Just to reiterate one more time, the basis states remain unchanged. It is the *sign* of the coefficient β that changes. The gate's constructor looks similar to the previous ones:

```
def PauliZ(d: int = 1) -> Operator:
    return Operator(np.array([[1.0, 0.0], [0.0, -1.0]])).kpow(d)
```

Together with the identity matrix, the Pauli matrices form a basis for the vector space of 2×2 Hermitian matrices. This means that all 2×2 Hermitian matrices can be constructed using a linear combination of Pauli matrices. They all have eigenvalues of 1.0 and -1.0. Pauli matrices are also involutory:

$$II = XX = YY = ZZ = I.$$

2.6.3 Rotations

Rotation operators are constructed via exponentiation of the Pauli matrices. Their impact can best be visualized as rotations around the Bloch sphere. We will discuss the Bloch sphere in more detail in Section 2.9. In short, every single qubit can be visualized as a point on a 3D sphere with a radius of 1.0 as shown in Figure 2.1. The $|0\rangle$ and $|1\rangle$ states are located at the north and south poles, respectively.

Applying a gate to a qubit moves its corresponding point from one surface point to another. This sphere lives in the 3D space, so there are x, y, and z axes with corresponding coordinates on the sphere. A qubit can reach any point on the sphere defined by spherical coordinates with $r = 1$ and the two angles θ and ϕ. The rotation around the z-axis $e^{i\phi}$ is called the *relative phase*.

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

We define rotations about the orthogonal axes x, y, and z, with help of the Pauli matrices as:

$$R_x(\theta) = e^{i\frac{\theta}{2}X},$$
$$R_y(\theta) = e^{i\frac{\theta}{2}Y},$$
$$R_z(\theta) = e^{i\frac{\theta}{2}Z}.$$

It can be shown that if an operator is involutory, then:

$$e^{i\theta A} = \cos(\theta)I + i\sin(\theta)A.$$

Proof If an operator function $f(A)$ has a power series expansion:

$$f(A) = c_0 I + c_1 A + c_2 A^2 + c_3 A^3 + \cdots.$$

then for the exponential function e^A we get:

$$f(A) = e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \frac{A^4}{4!} + \cdots.$$

For the function $e^{i\theta A}$ this is:

$$e^{i\theta A} = I + i\theta A - \frac{(\theta A)^2}{2!} - i\frac{(\theta A)^3}{3!} + \frac{(\theta A)^4}{4!} + \cdots.$$

If the operator is involutory and satisfies $A^2 = I$, then this becomes the following, which can be reordered into the Taylor series for $\sin(\cdot)$ and $\cos(\cdot)$:

$$\begin{aligned} e^{i\theta A} &= I + i\theta A - \frac{\theta^2 I}{2!} - i\frac{\theta^3 A}{3!} + \frac{\theta^4 I}{4!} + \cdots \\ &= \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \cdots\right) I + i\left(\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \cdots\right) A \\ &= \cos(\theta)I + i\sin(\theta)A. \end{aligned}$$

□

The Pauli matrices are involutory, with $II = XX = YY = ZZ = I$. Hence, we get rotation operators by multiplying out these expressions:

$$\begin{aligned} R_x(\theta) = e^{-i\frac{\theta}{2}X} &= \cos\left(\frac{\theta}{2}\right) I - i\sin\left(\frac{\theta}{2}\right) X \\ &= \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}, \end{aligned}$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

$$R_y(\theta) = e^{-i\frac{\theta}{2}Y} = \cos\left(\frac{\theta}{2}\right) I - i \sin\left(\frac{\theta}{2}\right) Y$$
$$= \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix},$$
$$R_z(\theta) = e^{-i\frac{\theta}{2}Z} = \cos\left(\frac{\theta}{2}\right) I - i \sin\left(\frac{\theta}{2}\right) Z$$
$$= \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}.$$

This also helps to explain why the Pauli Z-gate is called the phase-flip gate. Applying this gate will rotate the $|1\rangle$ part of a qubit around the z-axis by ϕ. With an angle $\phi = \pi$, the expression $e^{i\phi} = -1$, a rotation by 180°. This result is also famously known as *Euler's identity*:

$$e^{i\pi} = -1.$$

In general, rotations can be defined about *any* arbitrary axis $\hat{n} = (n_0, n_1, n_2)$:

$$R_{\hat{n}} = exp\left(-i\theta\hat{n}\frac{1}{2}\hat{\sigma}\right).$$

We learn more about general rotation and how to compute the axis and rotation angles in Section 6.15.3. For now, we can focus on the implementation of rotations about the standard Cartesian (x, y, z) axes:

```
# Cache Pauli matrices for performance reasons.
_PAULI_X = PauliX()
_PAULI_Y = PauliY()
_PAULI_Z = PauliZ()

def Rotation(v: np.ndarray, theta: float) -> np.ndarray:
    """Produce the single-qubit rotation operator."""

    v = np.asarray(v)
    if (v.shape != (3,) or not math.isclose(v @ v, 1) or
        not np.all(np.isreal(v))):
        raise ValueError('Rotation vector must be 3D real unit vector.')

    return np.cos(theta / 2) * Identity() - 1j * np.sin(theta / 2) * (
        v[0] * _PAULI_X + v[1] * _PAULI_Y + v[2] * _PAULI_Z)

def RotationX(theta: float) -> Operator:
    return Rotation([1., 0., 0.], theta)
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
def RotationY(theta: float) -> Operator:
    return Rotation([0., 1., 0.], theta)

def RotationZ(theta: float) -> Operator:
    return Rotation([0., 0., 1.], theta)
```

2.6.4 Phase Gate

The *phase gate*, also called the S-gate or Z90-gate, represents a phase of 90° around the z-axis for the $|1\rangle$ part of a qubit. This rotation is so common it gets its own name. In matrix form:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

It can be derived using Euler's formula for the angle $\phi = \pi/2$:

$$e^{i\phi} = \cos(\phi) + i\sin(\phi),$$
$$e^{i\pi/2} = \cos(\pi/2) + i\sin(\pi/2) = i.$$

Note that there is an important difference between this and the `RotationZ` gate. The S-gate only affects the second component of a qubit with its imaginary i, representing a 90° rotation. It leaves the first component of a qubit intact because of the 1.0 in the operator's upper left corner. In contrast, the `RotationZ` gate affects both components of a qubit. In code:

```
# Phase gate, also called S or Z90. Rotate by 90 deg around z-axis.
def Phase(d: int = 1) -> Operator:
    return Operator(np.array([[1.0, 0.0], [0.0, 1.0j]])).kpow(d)

# Phase gate is also called S-gate.
def Sgate(d: int = 1) -> Operator:
    return Phase(d)
```

To see the difference between the `RotationZ` gate and the S-gate, you can run a few simple experiments:

```
def test_rotation(self):
    rz = ops.RotationZ(math.pi)
    rz.dump('RotationZ pi/2')
    rs = ops.Sgate()
    rs.dump('S-gate')

    psi = state.qubit(random.random())
    psi.dump('Random state')
    ops.Sgate()(psi).dump('After applying S-gate')
    ops.RotationZ(math.pi)(psi).dump('After applying RotationZ')
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Which produces this output:

```
RotationZ pi/2 (1-qubits operator)
 -  -1.0j  -
 -         -   1.0j

S-gate (1-qubits operator)
 1.0       -
 -         -   1.0j

|0> 'Random state'
|0> (|0>):  ampl: +0.51+0.00j prob: 0.26 Phase:   0.0
|1> (|1>):  ampl: +0.86+0.00j prob: 0.74 Phase:   0.0
|0> 'After applying S-gate'
|0> (|0>):  ampl: +0.51+0.00j prob: 0.26 Phase:   0.0
|1> (|1>):  ampl: +0.00+0.86j prob: 0.74 Phase:  90.0
|0> 'After applying RotationZ'
|0> (|0>):  ampl: +0.00-0.51j prob: 0.26 Phase: -90.0
|1> (|1>):  ampl: +0.00+0.86j prob: 0.74 Phase:  90.0
```

See how the S-gate only affects the $|1\rangle$ component of the state, while the `RotationZ` gate affects both components?

We can spot a potential source of errors – the direction of rotations, especially when porting code from other infrastructures that might interpret angle directions differently. Fortunately, for much of this text, we are shielded from this problem. However, it may be one of the first things to look out for when results do not match expectations.

Finally, remember the Z-gate and how similar it looks to the phase gate? The relationship is easy to see – applying two phase gates, each effecting a rotation of $\pi/2$, yields a rotation of π, which we get from applying the Z-gate:

$$S^2 = SS = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = Z.$$

2.6.5 Flexible Phase Gates

In the context of the quantum Fourier transform, we will encounter gates performing phase rotations by fractions of π in two different forms.

The first gate is called the *discrete phase gate* (the R_k gate, or `Rk` gate), a generalization of the phase gate, performing rotations around the z-axis by fractional powers of 2, as in $2\pi/2^k$, e.g., 2π, π, $\pi/2$, $\pi/4$, $\pi/8$, and so on. Because rotation by 2π is a redundant operation, this gate makes sense only for $k > 0$.

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

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
# Rk is one of the rotation gates used in QFT.
def Rk(k: int, d: int = 1) -> Operator:
    return Operator(np.array([(1.0, 0.0),
            (0.0, cmath.exp(2.0 * cmath.pi * 1j / 2**k))])).kpow(d)
```

The next form is the `U1(lambda)` gate, also known as the phase shift or phase kick gate.

$$U_1(\lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix}.$$

It is similar to R_k, except arbitrary phase angles are allowed. In this text, we will only use power of 2 fractions of π. The implementation of the gate itself is straightforward:

```
def U1(lam: float, d: int = 1) -> Operator:
    return Operator(np.array([(1.0, 0.0),
                              (0.0, cmath.exp(1j * lam))])).kpow(d)
```

For integer powers of 2, the relationship between R_k and U_1 is the following:

$$R_k(0) = U_1\left(2\pi/2^0\right),$$
$$R_k(1) = U_1\left(2\pi/2^1\right),$$
$$R_k(2) = U_1\left(2\pi/2^2\right),$$
$$\vdots$$

You can verify this quickly:

```
def test_rk_u1(self):
    for i in range(10):
        u1 = ops.U1(2*math.pi / (2**i))
        rk = ops.Rk(i)
        self.assertTrue(u1.is_close(rk))
```

Some of the named gates are just special cases of R_k, in particular, the Identity-gate, Z-gate, S-gate, and T-gate (which we define in Section 2.6.6 below). This test code helps to clarify:

```
def test_rk(self):
    rk0 = ops.Rk(0)
    self.assertTrue(rk0.is_close(ops.Identity()))

    rk1 = ops.Rk(1)
    self.assertTrue(rk1.is_close(ops.PauliZ()))
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
    rk2 = ops.Rk(2)
    self.assertTrue(rk2.is_close(ops.Sgate()))

    rk3 = ops.Rk(3)
    self.assertTrue(rk3.is_close(ops.Tgate()))
```

2.6.6 Gates Square Roots

What is the square root of a classical NOT gate? There is no such thing. However, in quantum computing it is indeed possible to find a matrix $\sqrt{X}$. There are other interesting roots, some so common they also have their own names. As we will see later (Section 3.2.7), roots play an important role in constructing efficient two-qubit gates.

The root of the X-gate is the V-gate. V is unitary, with $VV^\dagger = I$, but also $V^2 = X$. It is defined in the following way:

$$V = \frac{1}{2}\begin{bmatrix} 1+i & 1-i \\ 1-i & 1+i \end{bmatrix}.$$

```
# V-gate, which is sqrt(X)
def Vgate(d: int = 1) -> Operator:
    return Operator(0.5 * np.array([(1+1j, 1-1j),
                                    (1-1j, 1+1j)])).kpow(d)
```

The root of a rotation is a rotation about the same axis, with the same direction, but by half the angle. This is obvious from the exponential form:

$$\sqrt{e^{i\phi}} = (e^{i\phi})^{\frac{1}{2}} = e^{i\phi/2}.$$

The root of the Phase gate (the S-gate) is called the T-gate. The S-gate represents a phase of 90° around the z-axis. Correspondingly, the T-gate is equivalent to a 45° phase around the z-axis.

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}.$$

```
def Tgate(d: int = 1) -> Operator:
    """T-gate is sqrt(S-gate)."""

    return Operator(
        np.array([[1.0, 0.0],
                  [0.0, cmath.exp(cmath.pi * 1j / 4)]])).kpow(d)
```

The root of the Y-gate has no special name (that we know of), but it is required later in the text, so we introduce it here as `Yroot`. It is defined as:

$$Y_{root} = \frac{1}{2}\begin{bmatrix} 1+i & -1-i \\ 1+i & 1+i \end{bmatrix}.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
def Yroot(d: int = 1) -> Operator:
    """Root of Y-gate."""

    return Operator(0.5 * np.array([(1+1j, -1-1j),
                                    (1+1j, 1+1j)])).kpow(d)
```

There are other interesting roots, but these are the main ones we will encounter in this text. We can test for the correct implementation of the roots with these snippets:

```
def test_t_gate(self):
    """Test that T^2 == S."""

    t = ops.Tgate()
    self.assertTrue(t(t).is_close(ops.Phase()))

def test_v_gate(self):
    """Test that V^2 == X."""

    v = ops.Vgate()
    self.assertTrue(v(v).is_close(ops.PauliX()))

def test_yroot_gate(self):
    """Test that Yroot^2 == Y."""

    yr = ops.Yroot()
    self.assertTrue(yr(yr).is_close(ops.PauliY()))
```

Finding a root in closed form can be quite cumbersome. In case of problems, you can simply use the `scipy` function `sqrtm()`. For this to work, `scipy` must be installed:

```
    from scipy.linalg import sqrtm
    [...]
    computed_yroot = sqrtm(ops.PauliY())
    self.assertTrue(ops.Yroot().is_close(computed_yroot))
```

2.6.7 Projection Operators

A projection operator for a given state, or *projector* for short, is the outer product of the state onto itself. This is also the definition of a density matrix, but projectors are usually constructed from basis states. The term projector comes from the simple fact that applying a basis state's projector to a given state extracts the amplitude of the basis state. The state is projected onto the basis state, similar to how the cosine is a projection of a 2-dimensional vector onto the x-axis. Let's see how this works. The projector is defined as:

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
def Projector(psi: state.State) -> Operator:
    """Construct projection operator for basis state."""

    return Operator(psi.density())
```

The projectors for the states $|0\rangle$ and $|1\rangle$ are:

$$P_{|0\rangle} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix},$$

$$P_{|1\rangle} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

Applying the projector for the $|0\rangle$ state to a random qubit yields the probability amplitude of the qubit being found in the $|0\rangle$ state (similar for the projector to the $|1\rangle$ state):

$$P_{|0\rangle}|\psi\rangle = |0\rangle\langle 0|(\alpha|0\rangle + \beta|1\rangle) = \alpha|0\rangle.$$

Projection operators are Hermitian, hence $P = P^\dagger$, but note that projection operators are *not* unitary and not reversible. If a projector operator's basis states are normalized, the projection operator is equal to its own square $P = P^2$; it is *idempotent*. This is a result we will use later in the section on measurement. Similar to basis states, two projection operators are *orthogonal* if and only if their product is 0, which means that for each state:

$$P_{|0\rangle} P_{|1\rangle}|\psi\rangle = \vec{0}.$$

To generalize, you can think of the outer product $|r\rangle\langle c|$ as a 2-dimensional index $[r, c]$ into a matrix. This is called the *outer product representation* of an operator.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} = a|0\rangle\langle 0| + b|0\rangle\langle 1| + c|1\rangle\langle 0| + d|1\rangle\langle 1|.$$

This also works for larger operators. For example, for this two-qubit operator U with just one nonzero element α:

$$U = \begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \alpha & 0 & 0 \end{pmatrix} \end{array}.$$

The outer product representation for the single nonzero element α in this operator would be $\alpha|11\rangle\langle 01|$, an index pattern of $|row\rangle\langle col|$. For derivations, this representation

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

can be more convenient than having to deal with full matrices. For example, to express the application of the X-gate to a qubit, we would write:

$$
\begin{aligned}
X &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|, \\
&X\big(\alpha\,|0\rangle + \beta\,|1\rangle\big) \\
&= \big(|0\rangle\langle 1| + |1\rangle\langle 0|\big)(\alpha\,|0\rangle + \beta\,|1\rangle) \\
&= |0\rangle\langle 1|\alpha\,|0\rangle + |0\rangle\langle 1|\beta\,|1\rangle + |1\rangle\langle 0|\alpha\,|0\rangle + |1\rangle\langle 0|\beta\,|1\rangle \\
&= \alpha|0\rangle \underbrace{\langle 1|0\rangle}_{=0} + \beta|0\rangle \underbrace{\langle 1|1\rangle}_{=1} + \alpha|1\rangle \underbrace{\langle 0|0\rangle}_{=1} + \beta|1\rangle \underbrace{\langle 0|1\rangle}_{=0} \\
&= \beta\,|0\rangle + \alpha|1\rangle.
\end{aligned}
$$

2.6.8 Hadamard Gate

Finally, we have arrived at the all-important Hadamard gate, which has the following form:

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

Let us apply this gate to $|0\rangle$ and $|1\rangle$ respectively:

$$
H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}}.
$$

$$
H|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}.
$$

Both results can be stated as the sum or difference of the $|0\rangle$ and $|1\rangle$ bases, scaled by $1/\sqrt{2}$. These states are so common they get their own symbolic names, $|+\rangle$ and $|-\rangle$:

$$
H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle,
$$

$$
H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle.
$$

The Hadamard gate puts a qubit into a superposition of the two basis states. This is why the Hadamard gate is so important – it generates superposition, one of the key ingredients of quantum computation. The states in superposition form an orthonormal basis as well, called the Hadamard basis. The basis states are $|+\rangle$ and $|-\rangle$, as indicated above.

For a general state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the Hadamard operator yields:

$$
\begin{aligned}
H|\psi\rangle &= H\big(\alpha|0\rangle + \beta|1\rangle\big) \\
&= \alpha H|0\rangle + \beta H|1\rangle
\end{aligned}
$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

$$
\begin{aligned}
&= \alpha\frac{|0\rangle + |1\rangle}{\sqrt{2}} + \beta\frac{|0\rangle - |1\rangle}{\sqrt{2}} \\
&= \alpha|+\rangle + \beta|-\rangle \\
&= \frac{\alpha+\beta}{\sqrt{2}}|0\rangle + \frac{\alpha-\beta}{\sqrt{2}}|1\rangle.
\end{aligned}
$$

```
def Hadamard(d: int = 1) -> Operator:
    return Operator(1 / np.sqrt(2) *
                    np.array([[1.0, 1.0], [1.0, -1.0]])).kpow(d)
```

Applying the Hadamard gate twice reverses the initial Hadamard gate. A Hadamard gate is its own inverse, $H = H^{-1}$, $HH = I$. It is involutory, just like the Pauli matrices:

$$
HH = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} = \frac{1}{2}\begin{bmatrix}2 & 0\\ 0 & 2\end{bmatrix} = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = I.
$$

A common operation is the application of Hadamard gates to several adjacent qubits. If all those qubits were in the $|0\rangle$ state, the resulting state becomes an *equal superposition* of the resulting states, all with the same amplitude of $\frac{1}{\sqrt{2^n}}$:

$$
H^{\otimes n}|0\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle.
$$

This construction is common for two and three qubits and used in many of the algorithms and examples. Let's spell it out explicitly:

$$
\begin{aligned}
&(H\otimes H)\big(|0\rangle\otimes|0\rangle\big) = \frac{1}{2}\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big), \\
&(H\otimes H\otimes H)\big(|0\rangle\otimes|0\rangle\otimes|0\rangle\big) \\
&= \frac{1}{\sqrt{2^3}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle\big) \\
&= \frac{1}{\sqrt{2^3}}\big(|0\rangle + |1\rangle + |2\rangle + |3\rangle + |4\rangle + |5\rangle + |6\rangle + |7\rangle\big) \\
&= \frac{1}{\sqrt{2^3}}\sum_{x=0}^{7}|x\rangle \quad = \frac{1}{\sqrt{2^3}}\sum_{x\in\{0,1\}^3}|x\rangle.
\end{aligned}
$$

Although we have seen single-qubit gates and learned how to construct multi-qubit states, a key ingredient to flexible computing is still missing. Where are the control flow constructs that are so common in classical computing and that seem essential for any type of algorithm? The quantum equivalents of those constructs are called *controlled gates*, which we discuss next.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

2.7 Controlled Gates

Quantum computing does not have classic control flow with branches around conditionally executed parts of the code. As described earlier, all qubits are active at all times. The quantum equivalent of control-dependent execution are *controlled* qubit gates.

These are gates that are always applied but only show effect under certain conditions. At least two qubits are involved: a controller qubit and a controlled qubit. Note that 2-qubit gates of this form cannot be decomposed into single-qubit gates.

REMARK *Before we continue, we have to agree on naming (which is hard). Shall we call a controlled not gate a, well, controlled not gate, a controlled-not gate, or Controlled-Not gate, or even a Controlled-Not-Gate?*

We will follow this convention: When we refer to an actual gate, or gate type, we will use the upper-case notation Controlled-Not*, sometimes followed by* gate*, but without a hyphen. For standard gates with single-letter names, we use a hyphen, as in* X-gate*. For gates with longer names we won't use a hyphen, such as* Swap gate *or* Hadamard gate*. In mathematical notation, gates are referred to by their symbolic names, such as* X, Y*, or* Z*.*

Let us explain the function of controlled gates by example. Consider how the following Controlled-Not matrix (abbreviated as CNOT, or CX) from qubit 0 to qubit 1 operates on all combinations of the $|0\rangle$ and $|1\rangle$ states.

$$CX_{0,1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Eagle-eyed readers will find the X-gate at the lower right side of this matrix and the identity matrix in the upper left. This can be misleading, as we show below for the controlled gate from 1 to 0. The important thing to note is that a Controlled-Not gate is a permutation matrix.

Applying this matrix to states $|00\rangle$ and $|01\rangle$ leaves the states intact:

$$CX_{0,1}\,|00\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = |00\rangle.$$

$$CX_{0,1}\,|01\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = |01\rangle.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Application to $|10\rangle$ *flips* the second qubit to a resulting state of $|11\rangle$.

$$CX_{0,1}\,|10\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = |11\rangle.$$

Application on $|11\rangle$ *flips* the second qubit to a resulting state of $|10\rangle$.

$$CX_{0,1}\,|11\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = |10\rangle.$$

The *CX* matrix flips the second qubit from $|0\rangle$ to $|1\rangle$, or from $|1\rangle$ to $|0\rangle$, but only if the first qubit is in state $|1\rangle$. The X-gate on the second qubit is *controlled* by the first qubit. Any 2×2 quantum gate can be controlled this way. We can have Controlled-Z gates, controlled rotations, or any other controlled 2×2 gates.

The *CX* gate is usually introduced, as we did here, by its effects on the $|0\rangle$ and $|1\rangle$ states of the second qubit. Only the amplitudes of the controlled qubit are being flipped. This is easy to see with the effects of the X-gate alone on a single qubit in superposition:

$$X|\psi\rangle = X\big(\alpha|0\rangle + \beta|1\rangle\big) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \beta|0\rangle + \alpha|1\rangle.$$

The *CX* matrix allows a first qubit to control an *adjacent* second qubit. What if the controller and controlled qubit are farther apart? The general way to construct a controlled unitary operator U with the help of projectors is the following:

$$CU_{0,1} = P_{|0\rangle} \otimes I + P_{|1\rangle} \otimes U. \tag{2.1}$$

Note that for a Controlled-Not gate $CX_{1,0}$ from 1 to 0, you cannot find the original X-gate or identity matrix in the operator. This matrix is still just a permutation matrix:

$$CX_{1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

If there are n qubits in between the controlling and controlled qubits, n identity matrices have to be tensored in between as well. If the index of the controlling qubit is larger than the index of the controlled qubit, the tensor products in Equation (2.1) need to be inverted. Here is an example with qubit 2 controlling gate U on qubit 0:

$$CU_{2,0} = I \otimes I \otimes P_{|0\rangle} + U \otimes I \otimes P_{|1\rangle}.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

The corresponding code is straightforward. We have to make sure that the right number of identity matrices are being added to pad the operator:

```
# Note on indices for controlled operators:
#
# The important aspects are direction and difference, not absolute
# values. In that regard, these are equivalent:
#  ControlledU(0, 3, U) == ControlledU(1, 4, U)
#  ControlledU(2, 0, U) == ControlledU(4, 2, U)
# We could have used -3 and +3, but felt this representation was
# more intuitive.
#
# Operator matrices are stored with all intermittent qubits
# (as Identities). When applying an operator, the starting qubit
# index can be specified.
def ControlledU(idx0: int, idx1: int, u: Operator) -> Operator:
  """Control qubit at idx1 via controlling qubit at idx0."""

  if idx0 == idx1:
    raise ValueError('Control and controlled qubit must not be equal.')

  p0 = Projector(state.zeros(1))
  p1 = Projector(state.ones(1))
  # space between qubits
  ifill = Identity(abs(idx1 - idx0) - 1)
  # 'width' of U in terms of Identity matrices
  ufill = Identity().kpow(u.nbits)

  if idx1 > idx0:
    if idx1 - idx0 > 1:
      op = p0 * ifill * ufill + p1 * ifill * u
    else:
      op = p0 * ufill + p1 * u
  else:
    if idx0 - idx1 > 1:
      op = ufill * ifill * p0 + u * ifill * p1
    else:
      op = ufill * p0 + u * p1
  return op
```

What is clear from this code is that operators larger than 2×2 can be controlled as well. We can construct Controlled-Controlled-... gates, which are required for most interesting algorithms.

This code makes one big operator matrix. This can be a problem in larger circuits, e.g., for a circuit with 20 qubits with qubit 0 controlling qubit 19 (or any other padded operator), the operator will be a matrix of size $(2^{20})^2$ * `sizeof(complex)`, which could be 8 or 16 terabytes[5] of memory. Building such a large matrix in memory and

[5] tebibytes, to be precise.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

applying it via matrix-vector multiplication is intractable. As this is how we express operators at this point, we are limited by the number of qubits we can experiment with. Fortunately, there are techniques to significantly improve scalability, which we will discuss detail in Chapter 4.

Also note that the controller and controlled qubits can be arbitrarily distant from each other in our simulations. In a real quantum computer, there are topological limitations on qubit interaction. Mapping an algorithm onto a concrete physical topology (IBM (2021b) shows several examples) introduces another interesting set of problems, which we will touch upon in Section 8.4.

2.7.1 Controlled-Not Gate

The Controlled-Not gates are so common that they deserve their own constructor function. We discussed the Controlled-Not (CNOT) gate at the beginning of Section 2.7. In code, its constructor looks like this:

```
def Cnot(idx0: int = 0, idx1: int = 1) -> Operator:
    """Controlled-Not between idx0 and idx1, controlled by |1>."""

    return ControlledU(idx0, idx1, PauliX())
```

The Controlled-Not-by-0 (CNOT0) gate is similar to the CNOT gate, except it is controlled by the $|0\rangle$ part of the controlling qubit. This can be accomplished by inserting an X-gate before and after the controlling qubit.

```
def Cnot0(idx0: int = 0, idx1: int = 1) -> Operator:
    """Controlled-Not between idx0 and idx1, controlled by |0>."""

    if idx1 > idx0:
        x2 = PauliX() * Identity(idx1 - idx0)
    else:
        x2 = Identity(idx0 - idx1) * PauliX()
    return x2 @ ControlledU(idx0, idx1, PauliX()) @ x2
```

Note that this construction to control a gate by $|0\rangle$ works for *any* target gate. We will see several examples of this in the later sections.

2.7.2 Controlled-Controlled-Not Gate

The full matrix construction works well in a nested fashion, extending the control to already-controlled gates. A double-controlled X-gate is also called the *Toffoli* gate, or CCX-gate for short.

This gate is interesting in classical computing because it can be shown that it is a universal gate – every classical logic function can be constructed using just this

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

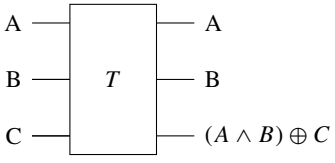


Figure 2.2 Block diagram for the Toffoli gate.

gate. Interestingly, this universality attribute does *not* hold in quantum computing. In quantum computing, there are only *sets* of universal gates (see also Section 6.15).

This is how the Toffoli gate works: if the first two inputs are $|1\rangle$ it flips the third qubit. This is often shown in form of a logic block diagram (with $\wedge$ as the logical AND), as in Figure 2.2.

In matrix form, we can describe it using block matrices, with 0_n as an $n \times n$ null matrix. Note that changing the indices of the controller and controlled qubits may destroy these patterns, but the matrix will still be a permutation matrix:

$$\begin{bmatrix} I_4 & 0_4 \\ 0_4 & CX \end{bmatrix} = \begin{bmatrix} I_2 & 0_2 & 0_2 & 0_2 \\ 0_2 & I_2 & 0_2 & 0_2 \\ 0_2 & 0_2 & I_2 & 0_2 \\ 0_2 & 0_2 & 0_2 & X \end{bmatrix}.$$

The constructor code is fairly straightforward and a good example of how to construct a double-controlled gate:

```
# Make Toffoli gate out of 2 controlled Cnot's.
#    idx1 and idx2 define the 'inner' cnot
#    idx0 defines the 'outer' cnot.
#
# For a Toffoli gate to control with qubit 5
# a Cnot from 4 and 1:
#    Toffoli(5, 4, 1)
#
def Toffoli(idx0: int, idx1: int, idx2: int) -> Operator:
    """Make a Toffoli gate."""

    cnot = Cnot(idx1, idx2)
    toffoli = ControlledU(idx0, idx1, cnot)
    return toffoli
```

We observe that because we are able to construct quantum Toffoli gates, and because Toffoli gates are classical universal gates, if follows that quantum computers are at least as capable as classical computers.

2.7.3 Swap Gate

The Swap gate is another important gate. It swaps the probability amplitudes of two qubits. In matrix representation, it looks like the following:

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

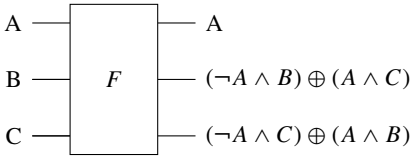


Figure 2.3 Block diagram for the Fredkin gate.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.2}$$

Our approach to construct controlled gates cannot produce this gate. However, it turns out that a sequence of three CNOT gates swaps the probability amplitudes of the basis states. For example, to swap qubit 0 and 1, you apply $CX_{10}\ CX_{01}\ CX_{10}$. This is analogous to classical computing, where a sequence of three XOR operations also swaps values. These techniques do not require additional temporary storage, such as a temporary variable or an additional helper qubit. There are other ways to construct Swap gates; some interesting examples are outlined in Gidney (2021b).

```
def Swap(idx0: int = 0, idx1: int = 1) -> Operator:
    """Swap qubits at idx0 and idx1 via combination of Cnot gates."""

    return Cnot(idx1, idx0) @ Cnot(idx0, idx1) @ Cnot(idx1, idx0)
```

2.7.4 Controlled Swap Gate

Like any other unitary operator, Swap gates can also be controlled. A double-controlled Swap gate is known as the *Fredkin gate*. Similar to the Toffoli gate, it is a universal gate in classical computing, but not in quantum computing. As a black box, it represents the logic shown in Figure 2.3, which may be a bit hard to reason about in isolation (with $\wedge$ as logical AND and $\neg$ as logical NOT).

The first physical quantum Fredkin gate was built relatively recently (Patel et al., 2016) and used to construct GHZ states, which we describe in Section 2.11.4.

2.8 Quantum Circuit Notation

We have learned the basics of qubits, states, operators, and gates, and how to combine them into larger circuits. We already hinted at a nice graphical way to visualize circuits. Here is how it works.

Qubits are drawn from top to bottom. Like the ordering we described earlier, the qubits are depicted from the "most significant" qubit to the "least significant" qubit. This can be confusing because you may naturally consider the top qubit as "qubit 0,"

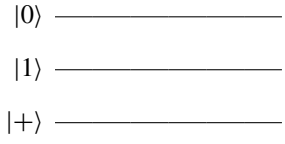
https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

which, in classical computing, denotes the least significant bit. In analytical equations, the top qubit will always be on the left of a state, such as the 1 in $|1000\rangle$.

Here is another way to visualize this ordering. If we imagine the column of qubits as a vector, transposing this vector will move the top qubit into the most significant (left-most) slot.

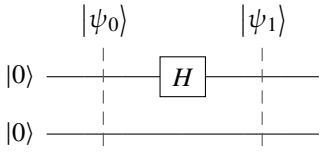
Graphically, the qubits' initial state is drawn to the left, and horizontal lines go to the right, indicating how the state changes over time as operators are applied. Again, note the absence of classical control flow. All gates are always active in a combined state, which could be a product state or an entangled state. Computation flows from left to right.

The initial state of three qubits appears like the following, with the initial state to the left of the circuit. It is conventional to always initialize qubits in state $|0\rangle$. However, because it is trivial to insert X-gates or Hadamard gates, we may take shortcuts and draw circuits as if these gates were present:



Also, note that the state of this circuit is the tensor product of the three qubits; it is a *combined* state (in the example, the state is still separable). It is tempting to reason that single, isolated qubits in the circuit diagram are in a particular state. However, the reality is that qubits are always in a combined state with other qubits, either as separable product states or entangled states.

Graphically, applying a Hadamard gate, or any other single-qubit operator, to the first qubit looks like the following. Before the operator is applied, the state is $|\psi_0\rangle = |0\rangle \otimes |0\rangle = [1,0,0,0]^T$. After the operator is applied, the state is $|\psi_1\rangle$. You can think about the state $|\psi_1\rangle$ as the tensor product of the top qubit being in superposition of $|0\rangle$ and $|1\rangle$ and the bottom qubit being in state $|0\rangle$.

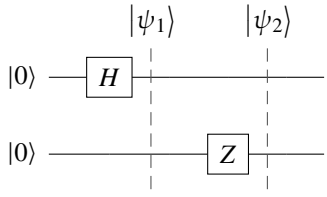


The initial state before the Hadamard gate is $|\psi_0\rangle = |0\rangle \otimes |0\rangle = |00\rangle$, the tensor product of the two $|0\rangle$ states. The Hadamard gate puts the top qubit into $1/\sqrt{2}(|0\rangle + |1\rangle) = |+\rangle$. As a result, the state $|\psi_1\rangle$ is the tensor product of the top qubit with the bottom qubit $|0\rangle$:

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

$$\left|\psi_1\right\rangle = |+\rangle \otimes |0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |0\rangle = \frac{1}{\sqrt{2}}\big(|00\rangle + |10\rangle\big). \tag{2.3}$$

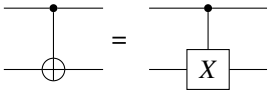
Applying a Z-gate after the Hadamard gate to qubit 1 results in this circuit:



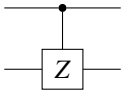
The fact that the Z-gate is to the right of the Hadamard gate indicates that this operator should be applied after the Hadamard gate. We can think of this as two separate gate applications – first the Hadamard H tensored with I, and then a second application of I tensored with Z. This is the equivalent of applying just one two-qubit operator O that has been constructed by multiplying the two tensor products (in reverse order of application, see Section 2.8.1):

$$O = (I \otimes Z)(H \otimes I).$$

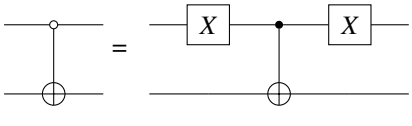
Controlled-X gates are indicated with a solid dot for the controller qubit and the addition-modulo-2 symbol $\oplus$ for the controlled qubit (not to be confused with the symbol for the tensor product $\otimes$). Addition-modulo-2 behaves like the binary XOR function. If the controlled qubit is $|0\rangle$ and we apply the X-gate, the gate becomes $|1\rangle$, as in $0 \oplus 1 = 1$. If the controlled bit is $|1\rangle$, applying the X-gate will turn it into $|0\rangle$, as in $1 \oplus 1 = 0$. The XOR gives the same truth table as addition-modulo-2. In some instances, we may still want to see an X-gate, but again, these two are identical:



Any single-qubit gate can be controlled this way, for example, the Z-gate:



The Controlled-Not-by-0 gate can be built by applying an X-gate before and after the controller. It is drawn with an empty circle on the controlling qubit:



https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

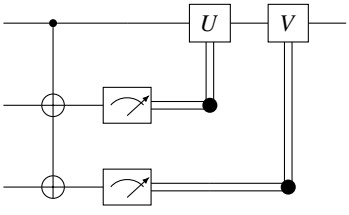
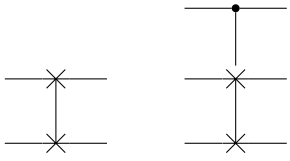
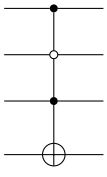


Figure 2.4 Measurement and flow of classical data after measurement, indicated with double lines.

Swap gates are marked with two connected $\times$ symbols, as in the circuit diagrams below. As discussed, like any other gate, Swap gates can also be controlled:



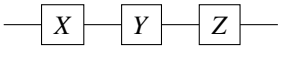
If a gate is controlled by more than one qubit, this can be indicated with multiple black or empty circles, depending on whether the gates are controlled by $|1\rangle$ or $|0\rangle$. In the example, qubits 0 and 2 must be $|1\rangle$ (have an amplitude for this base state), and qubit 1 must be $|0\rangle$ to activate the X-gate on qubit 3.



We will talk more about measurements in Section 2.15. Measurement gates produce real, classical values and are indicated with a meter symbol. Classical information flow is drawn with double lines. In the example in Figure 2.4, measurements are being made, and the real, classical measurement data may then be used to build or control other unitary gates, U and V in the example.[6]

2.8.1 Qubit Ordering Revisited

Let us reiterate the important point about ordering of gate applications. For a circuit like this following:



In code, we can apply the gate to the state from left to right, for example:

[6] All circuit diagrams in this book were created using the excellent LaTeX quantikz package.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
psi = state.zeros(1)
psi = ops.PauliX()(psi)
psi = ops.PauliY()(psi)
psi = ops.PauliZ()(psi)
```

Or we can construct a combined ZYX operator with function call syntax:

```
psi = state.zeros(1)
op = ops.PauliX(ops.PauliY(ops.PauliZ()))
psi = op(psi)
```

But if you want to write this as an explicit matrix multiplication, the order reverses. Take note of the parentheses – in Python, the function call operator has higher precedence than the matrix multiply operator:

```
psi = state.zeros(1)
psi = (ops.PauliZ() @ (ops.PauliY() @ ops.PauliX()))(psi)
```

In mathematical notation, a good rule to remember is that the operator closest to the bar of a ket (or bra) comes first when building an operator with explicit matrix multiplications:

$$\underbrace{X}_{\text{3rd}}\underbrace{Y}_{\text{2nd}}\underbrace{Z}_{\text{1st}}|\psi\rangle. \tag{2.4}$$

2.9 Bloch Sphere

We have seen several ways to describe states and gates, including Dirac notation, matrix notation, circuit notation, and code. You may prefer one over the other in your learning journey. Another representation is especially useful for visual learners: the Bloch sphere, named after the famous physicist Felix Bloch.

A single complex number can be drawn in a 2D polar coordinate system by specifying just a radius r and an angle ϕ to the x-axis. Typically, we think of this angle as counterclockwise. Complex numbers with radius $r = 1$ are restricted to a unit circle with radius 1.

A qubit is normalized because the probabilities of measuring a basis state must add up to 1.0. A qubit also has *two* complex amplitudes. Hence, two angles will suffice[7] to describe a qubit fully – it can be placed on the surface of a sphere with radius 1.0, the *unit sphere*.

In this representation, the $|0\rangle$ state is located at the north pole along the z-axis and the $|1\rangle$ state at the south pole of a sphere with radius 1. The $|+\rangle$ state points in the positive direction of the x-axis. Typically the x-axis is drawn as pointing out

[7] https://en.wikipedia.org/wiki/Qubit#Bloch_sphere_representation.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

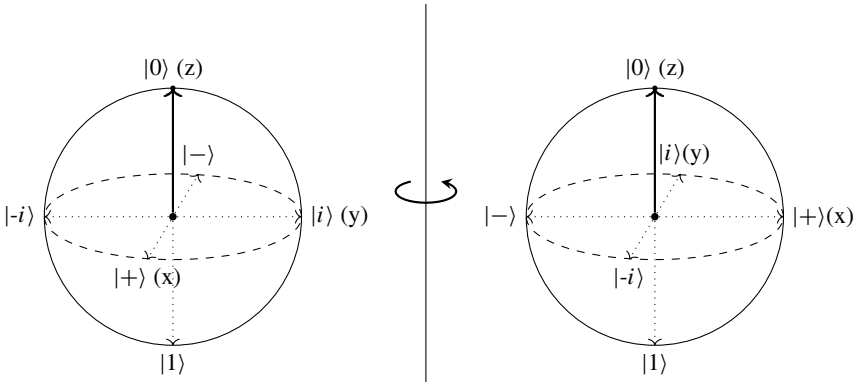


Figure 2.5 A Bloch sphere and the same sphere rotated counterclockwise by 90° about the z-axis.

of the page. The corresponding $|-\rangle$ state points in the negative x-direction into the page. The state $|i\rangle$ is on the Bloch sphere's equator on the positive y-axis, which is typically drawn to the right of the page. Correspondingly, $|\text{-}i\rangle$ is on the negative y-axis. The two spheres in Figure 2.5 are identical. The second sphere is rotated by 90° counterclockwise about the z-axis.

To see how to move about the sphere, let's start in state $|0\rangle$, the north pole of the sphere. Applying the Hadamard operator moves the state to $|+\rangle$, which is on the x-axis. This is shown in Figure 2.6a. Applying the Z-gate moves the arrow to point to state $|-\rangle$ on the negative x-axis. The relative phase is now $\pi/2$, as shown in Figure 2.6b.

It is obvious that, with rotations, you can reach any point on this sphere by using different paths or sequences of gate applications. Large rotations can also be broken down into equivalent sequences of smaller rotations.

This can also be demonstrated in code. In the example below, we start out with state $|1\rangle$ and apply the Hadamard gate to change the state to $|-\rangle$.

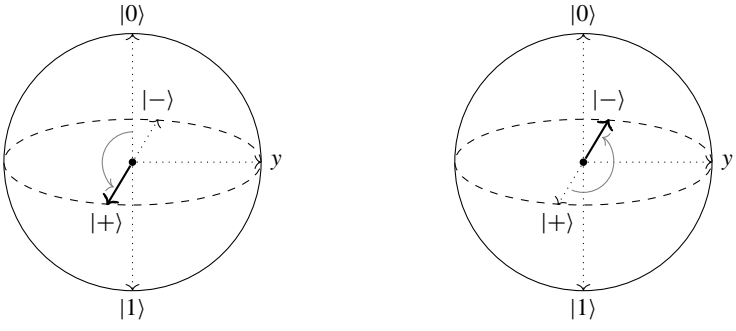
$$H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle.$$

Application of the X-gate changes the state to:

$$X\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) = \frac{|1\rangle - |0\rangle}{\sqrt{2}}.$$

Finally, we apply the Hadamard gate again, resulting in the state $-|1\rangle$. We should wonder about the minus sign and whether $-|1\rangle = |1\rangle$. We will show in Section 2.10 that we can *ignore* this minus sign because it is a *global phase*.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press



(a) Application of a H-gate (b) Application of a Z-gate

Figure 2.6 Rotating a state about the Bloch sphere.

$$
\begin{aligned}
H\left(\frac{|1\rangle - |0\rangle}{\sqrt{2}}\right) &= \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix}-1\\ 1\end{bmatrix} \\
&= \frac{1}{2}\begin{bmatrix}0\\ -2\end{bmatrix} = -1\begin{bmatrix}0\\ 1\end{bmatrix} \\
&= -|1\rangle.
\end{aligned}
$$

```
def basis_changes():
    """Explore basis changes via Hadamard."""

    # Generate [0, 1]
    psi = state.ones(1)

    # Hadamard on |1> will result in 1/sqrt(2) [1, -1]
    #   aka |->
    psi = ops.Hadamard()(psi)

    # Simple PauliX will result in 1/sqrt(2) [-1, 1]
    # which is -1 (1/sqrt(2) [1, -1]).
    # Note that this does not move the vector on the
    # Bloch sphere!
    psi = ops.PauliX()(psi)

    # Back to computational basis will result in -|1>.
    # Global phases can be ignored.
    psi = ops.Hadamard()(psi)
    if not np.allclose(psi[1], -1.0):
        raise AssertionError("Invalid basis change.")
```

As useful as the Bloch sphere can be, it may also lead to confusion. For example, the basis states $|0\rangle$ and $|1\rangle$ are orthogonal, but on the Bloch sphere they appear on opposite poles (similarly for $|+\rangle$, $|-\rangle$, and $|i\rangle$, $|\text{-}i\rangle$). It may be tempting to think

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

that classical rules of vector addition apply, but they do not: adding $|1\rangle$ and $-|1\rangle$ does not equal $|0\rangle$.

2.9.1 Coordinates

How do you compute the x, y, and z coordinates on the Bloch sphere for a given state $|\psi\rangle$? This is how we do it in two simple steps:

1. Compute the outer product of state $|\psi\rangle$ with itself to compute its density matrix $\rho = |\psi\rangle\langle\psi|$. We can use the convenience function `density()` for this, as described in Section 2.3.5.
2. Apply the helper function `density_to_cartesian(rho)`, shown below, which returns the corresponding x, y, z coordinates.

The function `density_to_cartesian(rho)` computes the Cartesian coordinates from a single-qubit density matrix. In the open-source repository, we add this function in file `lib/helper.py` .

How does the math[8] work? We stated in Section 2.6.2 that the Pauli matrices form a basis for the space of 2×2 Hermitian matrices. A density operator ρ is a 2×2 Hermitian matrix; we can write it as:

$$\rho = \frac{I + xX + yY + zZ}{2} = \frac{1}{2}\begin{bmatrix} 1+z & x-iy \\ x+iy & 1-z \end{bmatrix}. \tag{2.5}$$

If we think of ρ as a matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, then

$$2a = 1 + z,$$
$$2c = x + iy.$$

And correspondingly, $x = 2\,\mathrm{Re}(c)$, $y = 2\,\mathrm{Im}(c)$, and $z = 2a - 1$.

```
def density_to_cartesian(rho: np.ndarray) -> Tuple[float, float, float]:
  """Compute Bloch sphere coordinates from 2x2 density matrix."""

  a = rho[0, 0]
  c = rho[1, 0]
  x = 2.0 * c.real
  y = 2.0 * c.imag
  z = 2.0 * a - 1.0
  return np.real(x), np.real(y), np.real(z)
```

[8] https://quantumcomputing.stackexchange.com/a/17180/11582.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Here is a simple test to verify the results:

```
def test_bloch(self):
  psi = state.zeros(1)
  x, y, z = helper.density_to_cartesian(psi.density())
  self.assertEqual(x, 0.0)
  self.assertEqual(y, 0.0)
  self.assertEqual(z, 1.0)

  psi = ops.PauliX()(psi)
  x, y, z = helper.density_to_cartesian(psi.density())
  self.assertEqual(x, 0.0)
  self.assertEqual(y, 0.0)
  self.assertEqual(z, -1.0)

  psi = ops.Hadamard()(psi)
  x, y, z = helper.density_to_cartesian(psi.density())
  self.assertTrue(math.isclose(x, -1.0, abs_tol=1e-6))
  self.assertTrue(math.isclose(y, 0.0, abs_tol=1e-6))
  self.assertTrue(math.isclose(z, 0.0, abs_tol=1e-6))
```

Bloch spheres are only defined for single-qubit states. You can visualize an individual qubit's Bloch sphere in a many-qubit system by *tracing out* all the other qubits in the state. This is done with with the partial trace procedure, a helpful tool which we detail in Section 2.14.

2.10 Global Phase

What do we do with the minus sign on the final state after rotating the state about the Bloch sphere? Is $|1\rangle$ different from $-|1\rangle$? The Bloch sphere does not allow simple addition, $|1\rangle$ plus $-|1\rangle$ does *not* equal $|0\rangle$! The answer is that the minus sign in $-|1\rangle$ represents a *global phase*, a rotation by π. It has no *physical* meaning and can be ignored.

This is an important insight in quantum computing. A global phase is a complex coefficient $e^{i\phi}$ to a state with norm 1.0. Multiplying a state by a complex coefficient has no physical meaning because the expectation value of the state with or without the coefficient does not change. Physicists also call this *phase invariance*.

The *expectation value* for an operator A on state $|\psi\rangle$ is this expression (which we will develop in Section 2.15 on measurement):

$$\langle\psi|A|\psi\rangle.$$

With a global phase, this turns into:

$$\langle\psi|e^{-i\phi}Ae^{i\phi}|\psi\rangle = \langle\psi|e^{-i\phi}e^{i\phi}A|\psi\rangle = \langle\psi|A|\psi\rangle.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

The states cannot be distinguished by measurement.

We should contrast the global phase with a *relative phase*, which is the phase difference between the $|0\rangle$ and $|1\rangle$ parts of a qubit:

$$|\psi\rangle = \alpha\,|0\rangle + e^{i\phi}\beta\,|1\rangle.$$

2.11 Entanglement

The entanglement of two or more qubits is one of the most fascinating aspects of quantum physics. When two qubits (or systems) are entangled, it means that, on measurement, the results are strongly correlated, even when the states were physically separated, be it by a millimeter or across the universe! This is the effect that Albert Einstein famously called "spooky action at a distance." If we entangle two qubits in a specific way (described below), and qubit 0 is measured to be in state $|0\rangle$, qubit 1 will always be found in state $|0\rangle$ as well.

Why is this truly remarkable? What if we took two coins, placed them heads-up in two boxes, and shipped one of those boxes to Mars. When we open up the boxes, they will both show heads. So what's so special about the quantum case? In this example, coins have *hidden state*. We have placed them in the boxes *before* shipment, knowing which side to put on top in an initial, defined, nonprobabilistic state. We also know that this state will not change during shipment.

If there was hidden state in quantum mechanics, then the theory would be incomplete; the quantum mechanical wave functions would be insufficient to describe a physical state in full. This was the point that Einstein, Podolsky, and Rosen attempted to make in their famous *EPR paper* (Einstein et al., 1935).

However, a few decades later it was shown that there *cannot* be a hidden state in an entangled quantum system. A famous thought experiment, the Bell inequalities (Bell, 1964), proved this and was later experimentally confirmed.

Qubits collapse *probabilistically* during measurement to either $|0\rangle$ or $|1\rangle$.[9] This is equivalent to putting the coins in the boxes while they are twirling on their edges. Only when we open the boxes will the coins fall to one of their sides. Perfect coins would fall to each side 50% of the time. Similarly, if we prepared a qubit in the $|0\rangle$ state and applied a Hadamard gate to it, this qubit will measure either $|0\rangle$ or $|1\rangle$, with 50% probability for each outcome. The magic of quantum entanglement means that both qubits of an entangled pair will measure the same value, either $|0\rangle$ or $|1\rangle$, 100% of the time. This is equivalent to the coins falling to their same side, 100% of the time, on Earth and Mars!

There are profound philosophical arguments about entanglement, measurement, and what they tell us about the very nature of reality. Many of the greatest physicists of the last century have argued over this, for decades – Einstein, Schrödinger, Heisenberg,

[9] This is true as long as we measure in this basis. We talk about measurements in different bases in Section 6.11.3.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Bohr, and many others. These discussions are not settled to this day; there is no agreement. Many books and articles have been written about this topic, explaining it better than we would be able to do here. We are not even going to try. Instead, we accept the facts for what they are: rules we can exploit for computation.

This sentiment might put us in the camp of the Copenhagen interpretation of quantum mechanics (Faye, 2019). Ontology is a fancy term for questions like "What is?" or "What is the nature of reality?" The Copenhagen interpretation refuses to answer all ontological questions. To quote what David Mermin said about it (Mermin, 1989, p. 2): *If I were forced to sum up in one sentence what the Copenhagen interpretation says to me, it would be "Shut up and calculate!."* The key here is, of course, that progress can be made, even if the ontological questions remain unanswered.

2.11.1 Product States

Let us consider two-qubit systems. Constructing the tensor product between two qubits leads to a state where each qubit can still be described without reference to the other.

There is an intuitive (though not general) way to visualize this. The state can be expressed as the result of a tensor product with result $[a,b,c,d]^T$. If two states are *not* entangled, they are said to be in a *product state*; they are *separable*. This is the case if $ad = bc$. If the states *are* entangled, this identity will *not* hold.

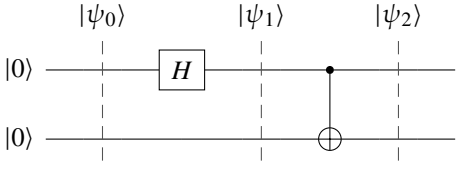
Proof As a quick proof, assume two qubits $q_0 = [i,k]^T$ and $q_1 = [m,n]^T$. Their Kronecker product is $q_0 \otimes q_1 = [im,in,km,kn]^T$. Multiplying the outer elements and the inner elements, corresponding to the $ad = bc$ form above, we see that

$$\underbrace{im}_{a}\underbrace{kn}_{d} = imkn = inkm = \underbrace{in}_{b}\underbrace{km}_{c}.$$

□

2.11.2 Entangler Circuit

The circuit below is the quintessential quantum entanglement circuit. We will see many uses of it in this text. Let us discuss in detail how the state changes as the gates are being applied.



The initial state $|\psi_0\rangle$, before the Hadamard gate, is the tensor product of the two $|0\rangle$ states, which is $|00\rangle$ with a state vector of $[1,0,0,0]^T$. The Hadamard gate puts the first qubit in superposition of $|0\rangle$ and $|1\rangle$. The state $|\psi_1\rangle$ after the Hadamard gate becomes

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

the tensor product of the superposition of the first qubit with the second qubit:

$$|\psi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}|0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle).$$

In code, we compute this with the following snippet. The `print` statement produces a state with nonzero entries at indices 0 and 2, corresponding to the states $|00\rangle$ and $|10\rangle$:

```
   psi = state.zeros(2)
   op = ops.Hadamard() * ops.Identity()
   psi = op(psi)
   print(psi)
>>
2-qubit state. Tensor:
[0.70710677+0.j 0.        +0.j 0.70710677+0.j 0.        +0.j]
```

Now we apply the Controlled-Not gate. The $|0\rangle$ part of the first qubit in superposition does not impact the second qubit; the $|00\rangle$ part remains unchanged. However, the $|1\rangle$ part of the superpositioned first qubit controls the second qubit and will flip it to $|1\rangle$, changing the $|10\rangle$ part to $|11\rangle$. The resulting state $|\psi_2\rangle$ after the Controlled-Not gate thus becomes:

$$|\psi_2\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}.$$

This state corresponds to this state vector:

$$|\psi_2\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix} = \begin{pmatrix}a\\b\\c\\d\end{pmatrix}.$$

This state is now entangled because the $ad = bc$ identity in the rule above does not hold: the product of elements 0 and 3 is $1/2$, but the product of elements 1 and 2 is 0. The state can no longer be expressed as a product state.

In code, we take the state `psi` we computed above and apply the Controlled-Not:

```
   psi = ops.Cnot(0, 1)(psi)
   print(psi)
```

This prints the entangled 2-qubit state, with elements 0 and 3 having values $1/\sqrt{2}$, corresponding to the binary indices of the $|00\rangle$ and $|11\rangle$ states:

```
2-qubit state. Tensor:
[0.70710677+0.j 0.        +0.j 0.        +0.j 0.70710677+0.j]
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Entanglement also means that now only states $|00\rangle$ and $|11\rangle$ can be measured. The other two basis states have a probability of 0.0 and cannot be measured. If qubit 0 is measured as $|0\rangle$, the other qubit will be measured as $|0\rangle$ as well, since the only nonzero probability state with a $|0\rangle$ as the first qubit is $|00\rangle$. Similar logic applies for state $|1\rangle$.

This explains the correlations, the spooky action at a distance, at least mathematically. The measurement results of the two qubits are 100% correlated. We don't know why this is, what physical mechanism facilitates this effect, or what reality is. But, at least for simple circuits and their respective matrices, we now have a means to express this unreal feeling reality.

2.11.3 Bell States

Bell states are named after the great physicist John Bell, who proved in 1964 using standard probability theory that entangled qubits cannot have hidden state or hidden information (Bell, 1964). This discovery was one of the defining moments for quantum mechanics.

We saw the first of four possible Bell states above, the one that was constructed with the entangler circuit with $|00\rangle$ as input. There are a total of four Bell states, resulting from the four inputs $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. We denote β_{xy} as the state resulting from inputs x and y:

$$|\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix}, \qquad |\beta_{01}\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\\1\\0\end{bmatrix},$$

$$|\beta_{10}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\-1\end{bmatrix}, \qquad |\beta_{11}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\\-1\\0\end{bmatrix}.$$

Here is the code to construct these states (in file `lib/bell.py` in the open-source repository):

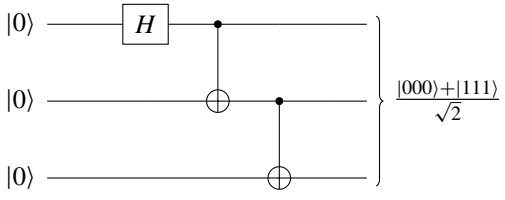
```
def bell_state(a: int, b: int) ->state.State:
 """Make one of the four bell states with a, b from {0,1}."""

 if a not in [0, 1] or b not in [0, 1]:
   raise ValueError('Bell state arguments are bits and must be 0 or 1.')
 psi = state.bitstring(a, b)
 psi = ops.Hadamard()(psi)
 return ops.Cnot()(psi)
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

2.11.4 GHZ States

A generalization of the Bell states are the n-qubit (maximally entangled) states of three or more qubits, named after Greenberger, Horne, and Zeilinger (GHZ) (Greenberger et al., 2008). They are constructed with this circuit, which propagates the superposition from the top qubit to all others via cascading Controlled-Not gates:



This construction can of course be extended to more than three qubits, generalizing to GHZ states $(|0000\ldots\rangle + |1111\ldots\rangle)/\sqrt{2}$. Only two possible states can be measured, each with probability $1/2$ (or amplitude $1/\sqrt{2}$). Note that instead of a cascade of Controlled-Not gates, we could just connect the Controlled-Not gates with qubit 0. In code:

```
def ghz_state(nbits: int) -> state.State:
 """Make a maximally entangled nbits state (GHZ State)."""

 # Simple construction via:
 #
 # |0> --- H --- o ---------     --- H --- o ----o----
 # |0> ----------X --- o --- or ----------X --- | ---
 # |0> ----------------X ---     ----------------X ---
 #    ...
 psi = state.zeros(nbits)
 psi = ops.Hadamard()(psi)
 for offset in range(nbits-1):
   psi = ops.Cnot(0, 1)(psi, offset)
 return psi
```

2.11.5 Maximal Entanglement

We used the term *maximally entangled* without explanation. Here is a simple definition:

A *maximally mixed* state is one where all probability amplitudes are the same. For example, applying Hadamard gates to all qubits in an n-qubit state initialized with $|0\rangle$ states leads to an equal superposition of all states.

$$(H \otimes H)|00\rangle = 1/2\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big)$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

We then define *maximally entangled* the following way. The partial trace allows reasoning about a subspace of a state, as explained in Section 2.14. It allows *tracing out* parts of a state. What is left is a reduced density matrix, representing the reduced state, the subspace. We call a state *maximally entangled* if the remaining reduced density matrices are maximally mixed, meaning their diagonal elements are all the same.

2.12 No-Cloning Theorem

There is another profound difference between classical computing and quantum computing. In classical computing, it is always possible to copy a bit, a byte, or any memory and copy it many times. In quantum computing, this is verboten – it is generally impossible to clone the state of a given qubit. This restriction is related to the topic of measurements and the fact that it is impossible to create a measurement device that does not impact (entangle with) a state. The inability to copy is expressed with the so-called No-Cloning Theorem (Wootters and Zurek, 1982).

THEOREM 2.1 *Given a general quantum state* $|\psi\rangle = |\phi\rangle|0\rangle$*, there cannot exist a unitary operator* U *such that* $U|\psi\rangle = |\phi\rangle|\phi\rangle$.

Proof Assume state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, then:

$$|\psi\rangle = |\phi\rangle|0\rangle = \big(\alpha|0\rangle + \beta|1\rangle\big)|0\rangle.$$

To clone $|\phi\rangle$ we need an operator U that performs this operation:

$$U|\phi\rangle|0\rangle = |\phi\rangle|\phi\rangle. \tag{2.6}$$

Applying U to $|\psi\rangle$ would thus lead to:

$$U|\phi\rangle|0\rangle = U\big(\alpha|0\rangle + \beta|1\rangle\big)|0\rangle = \alpha|00\rangle + \beta|11\rangle.$$

But now expand the right side of Equation (2.6):

$$\begin{aligned} U|\phi\rangle|0\rangle = |\phi\rangle|\phi\rangle &= \big(\alpha|0\rangle + \beta|1\rangle\big)\big(\alpha|0\rangle + \beta|1\rangle\big) \\ &= \alpha^2|00\rangle + \beta\alpha|10\rangle + \alpha\beta|01\rangle + \beta^2|11\rangle \\ &\neq \alpha|00\rangle + \beta|11\rangle. \end{aligned}$$

Hence, no such U exists. A general state cannot be cloned. □

General states can be *moved* but not *cloned*. Obviously this leads to interesting challenges in quantum algorithm design, and also the design of quantum programming languages.

Note, however, the special cases of $|0\rangle$ and $|1\rangle$. These states *can* be cloned. This is easy to see from the final form in the proof above. With one of a or b being 1.0 and the other being 0.0, only one term will remain:

$$|\phi\rangle|\phi\rangle = \alpha^2|00\rangle + \beta\alpha|10\rangle + \alpha\beta|01\rangle + \beta^2|11\rangle.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

With the result being one of:

$$\alpha^2|00\rangle = 1.0^2|00\rangle = |00\rangle,$$
$$\beta^2|11\rangle = 1.0^2|11\rangle = |11\rangle.$$

Matching the above result, with either $\alpha = 0$ or $\beta = 0$:

$$U|\phi\rangle|0\rangle = \alpha^2|00\rangle + \beta^2|11\rangle = \alpha|00\rangle + \beta|11\rangle.$$

2.13 Uncomputation

The question of logical reversibility of computation was raised by Bennett (1973). That paper was an answer to Landauer, who is also known for Landauer's principle (Landauer, 1973). The principle states that the *erasure* of information during computing must result in heat dissipation. The fact that today's CPUs run as hot as they do is a confirmation of this principle. Truly *reversible* computing would use almost no energy, but reversing a computation would also undo any obtained result. So the question was whether it was possible to construct a reversible circuit from which it was still possible to obtain an actual result. Given that quantum computing is reversible by definition, it would be utterly useless if we failed to answer this question. Fortunately, Bennet found an elegant construction to resolve this issue.

Bennet's paper is formal and based on a three-tape Turing machine. The proposed mechanism would compute a result, then *copy* the result before uncomputing the result via reverse computation of one of the Turing machine's tapes. The goal at the time was to mitigate heat dissipation. In quantum computation, our goal is to break undesirable entanglement to *ancilla qubits*. Bennet's approach works for both.

We mentioned ancilla qubits. Let's define the relevant terms:

- For constructions like the multi-controlled gate, as we will see in Section 3.2.8, we need *additional* qubits in order to properly perform the computation. You may think of these qubits as temporary qubits, or helper qubits, that play no essential role for the algorithm. They are the equivalent of compiler-allocated stack space mitigating classical register pressure. These qubits are called *ancilla* qubits, or *ancillae*.
- Ancilla qubits may start in state $|0\rangle$ and also end up in state $|0\rangle$ after a construction like the multi-controlled gate. In other scenarios, however, ancillae may remain entangled with a state, potentially destroying a desired result. In this case, we call these ancillae *junk qubits*, or simply *junk*.

The typical structure of a quantum computation looks like that shown in Figure 2.7. All quantum gates are unitary, so we pretend we packed them all up in one giant unitary operator U_f. There is the input state $|x\rangle$ and some ancillae qubits, all initialized to $|0\rangle$. The result of the computation will be $f(|x\rangle)$ and some left over ancillae, which are now *junk*; they serve no purpose, they just hang around, intent on messing up

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

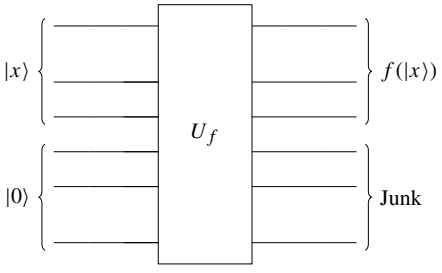


Figure 2.7 Typical structure of a quantum computation.

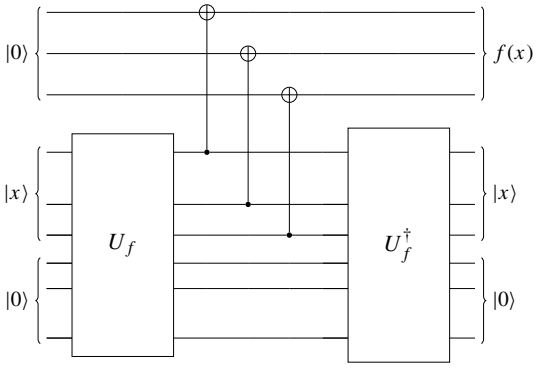


Figure 2.8 Computation, fan-out, uncomputation.

our results. The problem is that the junk qubits may still be entangled with the result, nullifying the intended effects of quantum interference, which quantum algorithms are based upon. We will not be exposed to this problem until Section 6.6, where we have to solve this problem for the order finding part of Shor's algorithm.

Here is the procedure, as shown in Figure 2.8. After computing a solution, we apply the inverse unitary operations to undo the computation completely. We can either build a giant combined unitary adjoint operator, or, if we have constructed a circuit from individual gates, we apply the inverses of the gates in reverse order. This works because operators are unitary and $U^\dagger U = I$.

The problem now is that we lost the result $f(|x\rangle)$ that we were trying to compute. Here is the "trick" to work around the problem, which is similar to Bennet's recipe. After computation, but before uncomputation, we connect the result qubits out to another quantum register via Controlled-Not gates, as shown in Figure 2.8.

With this circuit, the result of $f(|x\rangle)$ will be in the upper register, and the state of the other registers will be restored to their original state, eliminating all unwanted entanglement.

Why does this work at all? We start in the state composed of an input state $|x\rangle$ and a working register initialized with $|0\rangle$. The first U_f transforms the initial state $|x\rangle|0\rangle$ into $f(|x\rangle)g(|0\rangle)$. We add the ancillae register at the top, building a product state $|0\rangle f(|x\rangle)g(|0\rangle)$, and CNOT the register holding the result to get $f(|x\rangle)f(|x\rangle)g(|0\rangle)$.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

This does *not* violate the no-cloning theorem; the two registers cannot be measured independently and give the same result. We apply $U_f^\dagger$ to the bottom two registers to uncompute U_f and obtain $f(|0\rangle)|x\rangle|0\rangle$. The final result is in the top register; the bottom registers have been successfully restored.

2.14 Reduced Density Matrix and Partial Trace

For debugging and analysis purposes, it is often desirable to inspect a subsystem of a given state. For this, we use what is called a *reduced density operator*, which we can derive with help of a procedure called a *partial trace*. This section might be a bit confusing to novices, but the technique discussed here is not used until much later in the text (see, for example, Section 6.2). You may choose to come back to it later as well.

We compute a partial trace from a state's density matrix. We briefly mentioned that the whole theory of quantum computing can be expressed using density matrices, but for simplicity, we did not elaborate on this representation. For the partial trace, however, we need to revisit this topic briefly. Two concepts are important.

First, a density matrix of a state is the outer product of the state with itself. This representation describes the whole system, including its potential entanglement with the environment. This gives rise to the notion of *pure* states, which are defined by the state vector alone, and *mixed* states, which may be entangled with something else, or are a statistical mixture of states.

Second, to apply an operator to a state as a density matrix, it applies from left and right. In state representation, we use $|\psi'\rangle = U|\psi\rangle$. To achieve the same state evolution in the density matrix representation, we apply $\rho' = U\rho U^\dagger$.

Assume we have a state $|\psi\rangle$ that is a combination of A and B with density operator $\rho^{AB} = |AB\rangle\langle AB|$. The reduced density operator ρ_A for subspace A is defined as $\rho_A = \mathrm{tr}_B(\rho^{AB})$, where tr_B is the partial trace over system B. This means that ρ_A contains the original state, but with B removed. It has been "traced out."

We want to have a mathematical tool that extracts a subspace out of the density operator. In terms of measurement, we only want to compute the trace over a subset of the full density operator, e.g., the subset describing either A or B. If ρ is the density matrix for the combined state, we want to compute either of the following:

$$\rho_B = \mathrm{tr}_A(\rho^{AB}) = \langle 0_A|\rho|0_A\rangle + \langle 1_A|\rho|1_A\rangle,$$
$$\rho_A = \mathrm{tr}_B(\rho^{AB}) = \langle 0_B|\rho|0_B\rangle + \langle 1_B|\rho|1_B\rangle.$$

$|0_A\rangle$ and $|1_A\rangle$ are the projectors on the basis states $|0\rangle$ and $|1\rangle$. Projectors are Hermitian, hence $P = P^\dagger$. Note that the partial trace is a *dimension reducing* operation. As an example, let us assume we have a system of two qubits A and B (at indices 0 and 1) with a 4×4 density matrix, and we want to trace out qubit 0.

We construct $|0_A\rangle$ and $|1_A\rangle$ by tensoring the $|0\rangle$ and $|1\rangle$ states with an identity matrix I, resulting in a matrix of size 4×2. The order depends on which specific qubit we intend to trace out. With an approach similar to that used for gate applications, we pad the $|0\rangle$ and $|1\rangle$ states with identity matrices from left and right. In our example,

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

we want to trace out qubit 0, so the $|0\rangle$ and $|1\rangle$ states come first and are tensored with an identity matrix:

$$|0_A\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} = \begin{pmatrix}1 & 0\\0 & 1\\0 & 0\\0 & 0\end{pmatrix}, \quad |1_A\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} = \begin{pmatrix}0 & 0\\0 & 0\\1 & 0\\0 & 1\end{pmatrix}.$$

Multiplying the 4×4 density to the right with a 4×2 matrix results in a 4×2 matrix. Multiplying this matrix from the left with a (now transposed) 2×4 matrix results in a 2×2 matrix.

This definition of the partial trace is sufficient for implementation. The following code is written in full matrix form and will only scale to a small number of qubits.

```
def TraceOutSingle(rho: Operator, index: int) -> Operator:
  """Trace out single qubit from density matrix."""

  nbits = int(math.log2(rho.shape[0]))
  if index > nbits:
    raise AssertionError(
        'Error in TraceOutSingle, invalid index (>nbits).')

  eye = Identity()
  zero = Operator(np.array([1.0, 0.0]))
  one = Operator(np.array([0.0, 1.0]))

  p0 = p1 = tensor.Tensor(1.0)
  for idx in range(nbits):
    if idx == index:
      p0 = p0 * zero
      p1 = p1 * one
    else:
      p0 = p0 * eye
      p1 = p1 * eye

  rho0 = p0 @ rho
  rho0 = rho0 @ p0.transpose()
  rho1 = p1 @ rho
  rho1 = rho1 @ p1.transpose()
  rho_reduced = rho0 + rho1
  return rho_reduced
```

If we have a state of n qubits and are interested in the state of just one of the qubits, we have to trace out *all other* qubits. Here is a convenience function for this:

```
def TraceOut(rho: Operator, index_set: List[int]) -> Operator:
  """Trace out multiple qubits from density matrix."""

  for idx, val in enumerate(index_set):
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
    nbits = int(math.log2(rho.shape[0]))
    if val > nbits:
      raise AssertionError('Error TraceOut, invalid index (>nbits).')
    rho = TraceOutSingle(rho, val)

    # Tracing out a bit means that rho is now 1 bit smaller, the
    # indices right to the traced out qubit need to shift left by 1.
    # Example, to trace out bits 2, 4:
    # Before:
    #    qubit 0  1  2  3  4  5
    #          a  b  c  d  e  f
    # Trace out 2:
    #    qubit 0  1 <-  3  4  5
    #    qubit 0  1  2  3  4
    #          a  b  d  e  f
    # Trace out 4 (is now 3)
    #    qubit 0  1  2  <-  4
    #    qubit 0  1  2  3
    #          a  b  d  f
    for i in range(idx+1, len(index_set)):
      index_set[i] = index_set[i] - 1
  return rho
```

2.14.1 Experiments

Now let us see this procedure in action. We start by producing a state from two well-defined qubits.

```
q0 = state.qubit(alpha=0.5)
q1 = state.qubit(alpha=0.8660254)
psi = q0 * q1
```

Tracing out one qubit should leave the other in the resulting density matrix, with matrix element (0,0) holding the value $|\alpha|^2$ and matrix element (1, 1) holding the value $|\beta|^2$. Remember that the density matrix is the outer product of a state vector, with:

$$\begin{pmatrix} a \\ b \end{pmatrix} \begin{pmatrix} a^* & b^* \end{pmatrix} = \begin{pmatrix} aa^* & ab^* \\ ba^* & bb^* \end{pmatrix}.$$

We have seen earlier that

$$\mathrm{tr}(|x\rangle\langle y|) = \sum_{i=0}^{n-1} x_i y_i^* = \langle y|x\rangle. \tag{2.7}$$

For an outer product of a state vector, we know that the trace must be 1.0, as the probabilities add up to 1.0. Correspondingly, density matrices also must have a trace of 1.0. This is confirmed above; the diagonal elements are the squares of the norms of the probability amplitudes.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

Tracing out qubit 1 should result in a value of $0.5 * 0.5 = 0.25$ in matrix element (0, 0), which is the norm squared of qubit 0's α value of 0.5:

```
reduced = ops.TraceOut(psi.density(), [1])
self.assertTrue(math.isclose(np.real(np.trace(reduced)), 1.0))
self.assertTrue(math.isclose(np.real(reduced[0, 0]),
                             0.25, abs_tol=1e-6))
self.assertTrue(math.isclose(np.real(reduced[1, 1]),
                             0.75, abs_tol=1e-6))
```

Tracing out qubit 0 should leave $0.8660254^2 = 0.75$ in matrix element (0, 0):

```
reduced = ops.TraceOut(psi.density(), [0])
self.assertTrue(math.isclose(np.real(np.trace(reduced)), 1.0))
self.assertTrue(math.isclose(np.real(reduced[0, 0]),
                             0.75, abs_tol=1e-6))
self.assertTrue(math.isclose(np.real(reduced[1, 1]),
                             0.25, abs_tol=1e-6))
```

This becomes interesting for entangled states. Take, for example, the first Bell state. If we compute the density matrix and square it, the trace of the squared matrix is 1.0. If we trace out qubit 0:

```
psi = bell.bell_state(0, 0)
reduced = ops.TraceOut(psi.density(), [0])
self.assertTrue(math.isclose(np.real(np.trace(reduced)),
                             1.0, abs_tol=1e-6))
self.assertTrue(math.isclose(np.real(reduced[0, 0]),
                             0.5, abs_tol=1e-6))
self.assertTrue(math.isclose(np.real(reduced[1, 1]),
                             0.5, abs_tol=1e-6))
```

We see that the result is $I/2$, the diagonals are all the same, the state was maximally entangled. The trace of the squared, reduced density matrix is 0.5. We mentioned pure and mixed states above. The trace operation gives us a mathematical definition:

$$\text{tr}(\rho^2) < 1 : \text{Mixed State},$$
$$\text{tr}(\rho^2) = 1 : \text{Pure State}.$$

The result for the partial trace on the Bell state shows that the remaining qubit is in a mixed state:

$$\text{tr}\left((I/2)^2\right) = 0.5 < 1.$$

The joint state of the two qubits, entangled or not, is a pure state; it is known exactly, it is not entangled further with the environment. However, looking at the individual qubits of the entangled Bell state, we find that those are in a mixed state – we do not have full knowledge of their state.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

2.15 Measurement

We have arrived at the end of the introductory section. What remains is to discuss measurements. This is a complex subject with many subtleties and layered theories. Here, we stick to the minimum – projective measurements.

2.15.1 Postulates of Quantum Mechanics

There are five postulates of quantum mechanics. Depending on the context, you may find them presented in a different order, with a different focus and rigor. In keeping with the spirit of our text, we present them here in an informal way that conveys just enough information to understand the essence of the postulates.

1. The state of a system is represented by a ket, which is a unit vector of complex numbers representing probability amplitudes.
2. A state evolves as the result of unitary operators operating on the state, $|\psi'\rangle = U|\psi\rangle$. This is derived from the time-independent Schrödinger equation. To describe the evolution of a system in continuous time, this postulate is expressed with the time-dependent Schrödinger equation (which we mostly ignore in this text).
3. Measurement means collapsing the probability function to a singular, measurable value, which is a real eigenvalue of the Hermitian measurement operator. This sounds scarier than it is, and it is the focus of this section.
4. The probability amplitudes and corresponding probabilities determine the likelihood of a specific measurement result.
5. After measurement, the state *collapses* to the result of the measurement. This is called the *Born rule*. We explain the implications of this postulate, in particular, the need for renormalization.

In the preceding chapters, we already *captured* postulates one and two by expressing states as full state vectors and showing how to apply unitary operators. By means of probability amplitudes, we have implicitly used postulate four and, to some degree, postulate five as well. In this section, we will focus on postulate three, measurements.

An important thing to note is that the postulates are postulates, not standard physical laws. As noted above, they are also the subject of almost a century's worth of scientific disputes and philosophical interpretation. See, for example Einstein et al. (1935), Bell (1964), Norsen (2017), Faye (2019), and Ghirardi and Bassi (2020), and many more. Nevertheless, as indicated before, we avoid philosophy and focus on how the postulates enable interesting forms of computation.

2.15.2 Projective Measurements

The class of projective measurements is easy to understand and the only method we will use in this text. Given a system that is in one of two states, e.g., an atom with an

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

energy state high or low, the idea behind making a projective measurement is to simply determine which of those two states the atom was in. Qubits are in superposition. We may wonder whether it is in the $|0\rangle$ state or the $|1\rangle$ state, but measurement can only return one of these two states, which it does with a given probability. We know that after measurement, according to the Born rule, the state *collapses* to the measured state (postulate 5). It is now either $|0\rangle$ or $|1\rangle$ and will be found in this state for all future measurements.

Why is this called a projective measurement? In the section on single-qubit operators, we already learned about projection operators.

$$P_{|0\rangle} = |0\rangle\langle 0| = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix},$$
$$P_{|1\rangle} = |1\rangle\langle 1| = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

Applying a projector to a qubit "extracts" a subspace. For example, for $P_{|0\rangle}$:

$$\begin{aligned} P_{|0\rangle}|\psi\rangle &= |0\rangle\langle 0|(\alpha|0\rangle + \beta|1\rangle) \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \end{bmatrix} \\ &= \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \alpha|0\rangle. \end{aligned}$$

We can say that the probability $Pr(i)$ of finding the ith basis state on measurement is the following, where we square the norm of the probability amplitude, as stated in the fourth postulate:

$$Pr(i) = \left(P_{|i\rangle}|\psi\rangle\right)^2.$$

Following Equation (1.2) to compute the vector norm and Equation (1.10) to compute the Hermitian adjoint of an expression, we have

$$\begin{aligned} Pr(i) &= \left(P_{|i\rangle}|\psi\rangle\right)^\dagger \left(P_{|i\rangle}|\psi\rangle\right) \\ &= \langle\psi|P_{|i\rangle}^\dagger P_{|i\rangle}|\psi\rangle. \end{aligned}$$

Projectors are Hermitian and hence equal to their adjoint:

$$\begin{aligned} Pr(i) &= \langle\psi|P_{|i\rangle} P_{|i\rangle}|\psi\rangle \\ &= \langle\psi|P_{|i\rangle}^2|\psi\rangle. \end{aligned}$$

We also know that for projectors of normalized basis vectors (which have 1s on the diagonal):

$$P_{|i\rangle}^2 = P_{|i\rangle},$$

which results in this final form for the probability:

$$Pr(i) = \langle\psi|P_{|i\rangle}|\psi\rangle.$$

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

The term $\langle\psi|P_{|i\rangle}|\psi\rangle$ is also called the *expectation value* of operator $P_{|i\rangle}$, the quantum equivalent of the *average* of $P_{|i\rangle}$. It can be written as $[P_{|i\rangle}]$.

We know from the section on the trace of a matrix (Equation (1.19)) that

$$\mathrm{tr}\big(|x\rangle\langle y|\big) = \sum_{i=0}^{n-1} x_i y_i^* = \langle y|x\rangle. \tag{2.8}$$

By rearranging terms and using Equation (2.8), we finally arrive at the form we will use in our code:

$$Pr(i) = \langle\psi|P_{|i\rangle}|\psi\rangle = \mathrm{tr}\big(P_{|i\rangle}|\psi\rangle\langle\psi|\big). \tag{2.9}$$

You can understand this form intuitively. The density matrix of the state $|\psi\rangle\langle\psi|$ has the probabilities $Pr(i)$ for each basis state $|x_i\rangle$ on the diagonal, as shown in Section 2.3.5. The projector zeros out all diagonal elements that are not covered by the projector's basis state. What remains on the diagonal are the probabilities of states matching the projector. The trace then adds up all these remaining probabilities off the diagonal.

After measurement, the state collapses to the measured result. Basis states that do not agree with the measured qubit values get a resulting probability of 0.0 and "disappear." As a result, the remaining states' probabilities no longer add up to 1.0 and need to be renormalized, which we achieve with the complicated-looking expression (no worries, in code, this will look quite simple):

$$|\psi\rangle = \frac{P_{|i\rangle}|\psi\rangle}{\sqrt{\langle\psi|P_{|i\rangle}|\psi\rangle}}. \tag{2.10}$$

As an example, let us assume we have this state:

$$|\psi\rangle = 1/2\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big).$$

Each of the four basis states has equal probability $\left(\frac{1}{2}\right)^2 = 1/4$ of being measured. Let us further assume that qubit 0 is being measured as $|0\rangle$. This means that the only choices for the final, full state to be measured are $|00\rangle$ or $|01\rangle$. The first qubit is "fixed" at $|0\rangle$ after measurement. This means that the states where qubit 0 is $|1\rangle$ now have a 0% probability of ever being measured. The state collapsed to this unnormalized state:

$$|\psi\rangle_{(\neq|1|)} = 1/2\big(|00\rangle + |01\rangle\big) + 0.0\big(|10\rangle + |11\rangle\big).$$

But in this form, the squares of the probability amplitudes no longer add up to 1.0. We must renormalize the state following Equation (2.10) and divide by the square root of the expectation value (which was $1/2$) to get:

$$|\psi\rangle = 1/\sqrt{2}\big(|00\rangle + |01\rangle\big).$$

This step might be surprising. How does Nature know when and if to normalize? Given that we're adhering to the Copenhagen interpretation and decided to "Shut up and compute!," a possible answer is that the need for re-normalization is simply a remnant of the mathematical framework, nothing more, nothing less.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

2.15.3 Implementation

The function to measure a specific qubit has the following parameters:

- The state to measure is passed as parameter `psi`.
- Which qubit to measure, indexed from the top/left, with parameter `idx`.
- Whether to measure the probability that the state collapses to $|0\rangle$ or $|1\rangle$ is controlled with parameter `tostate`.
- After measurement, whether the state should collapse or not is controlled by parameter `collapse`. In the physical world, measurement destroys superposition, but in our simulation, we can just take a peek-a-boo at the probabilities without affecting the superposition of states.

The way this function is written, if we measure and collapse to state $|0\rangle$, the state will be made to collapse to this state, *independent* of the probabilities. There are other ways to implement this, e.g., by selecting the measurement result based on the probabilities. At this early point in our exploration, the ability to force a result works quite well; it makes debugging easier. Care must be taken, though, never to force the state to collapse to a result with probability 0. This would lead to a division by 0 and likely very confusing subsequent measurement results.

The function returns two values: the probability of measuring the desired qubit state and a state. This state is either the post-measurement collapsed state, if `collapse` was set to `True`, or the unmodified state otherwise. Here is the implementation. The function first computes the density matrix and the padded operator around the projection operator:

```
def Measure(psi: state.State, idx: int,
            tostate:int=0, collapse:bool=True) -> (float, state.State):
 """Measure a qubit via a projector on the density matrix."""

 # Compute probability of qubit(idx) to be in state 0 / 1.
 rho = psi.density()
 op = Projector(state.zero) if tostate = 0 else Projector(state.one)

 # Construct full matrix to apply to density matrix:
 if idx > 0:
   op = Identity().kpow(idx) * op
 if idx < psi.nbits - 1:
   op = op * Identity().kpow(psi.nbits - idx -1)
```

The probability is computed from a trace over the matrix resulting from multiplication of the padded projection operator with the density matrix, as in Equation 2.9:

```
 # Probability is the trace.
 prob0 = np.trace(np.matmul(op, rho))
```

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

If state collapse is required, we update the state and renormalize it before returning the updated (or unmodified) probability and state.

```
  # Collapse state (don't forget to normalize if norm != 0)
  if collapse:
    mvmul = np.dot(op, psi)
    divisor = np.real(np.linalg.norm(mvmul))
    if divisor > 1e-10:
      normed = mvmul / divisor
    else:
       raise AssertionError(
               'Measure() collapsed to 0.0 probability state.')
    return np.real(prob0), state.State(normed)

  # Return original state, enable chaining.
  return np.real(prob0), psi
```

And just to clarify one more time, the measurement operators are projectors. They are Hermitian with eigenvalues 0 and 1, and eigenvectors $|0\rangle$ and $|1\rangle$. A measurement will produce $|0\rangle$ or $|1\rangle$, corresponding to the basis states' probabilities. Measurement will not measure, say, a value of 0.75. It will measure one of the two basis states with probability 0.75. This can be a source of confusion for beginners – in the real world we have to measure several times to find the probabilities with statistical significance.

2.15.4 Examples

Let us look at a handful of examples to see measurements in action. In the first example, let us create a 4-qubit state and look at the probabilities:

```
    psi = state.bitstring(1, 0, 1, 0)
    psi.dump()
>>
    |1010> (|10>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
```

There is only one state with nonzero probabilities. If we measure the second qubit to be 0, which it is:

```
    p0, _ = ops.Measure(psi, 1)
    print(p0)
>>
    1.0
```

But if we tried to measure this second qubit to be 1, which it cannot be, we will get an error, as expected:

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press

```
    p0, _ = ops.Measure(psi, 1, tostate=1)
    print(p0)
>>
    AssertionError: Measure() collapsed to 0.0 probability state
```

Here is an example with a collapsing measurement. Let us create a Bell state:

```
    psi = bell.bell_state(0, 0)
    psi.dump()
>>
    |00> (|0>):  ampl: +0.71+0.00j prob: 0.50 Phase:   0.0
    |11> (|3>):  ampl: +0.71+0.00j prob: 0.50 Phase:   0.0
```

The state has only two possible measurement outcomes, $|00\rangle$ and $|11\rangle$. Let us measure the first qubit to be $|0\rangle$ without collapsing the state:

```
    psi = bell.bell_state(0, 0)
    p0, _ = ops.Measure(psi, 0, 0, collapse=False)
    print('Probability: ', p0)
    psi.dump()
>>
    Probability:  0.49999997
    |00> (|0>):  ampl: +0.71+0.00j prob: 0.50 Phase:   0.0
    |11> (|3>):  ampl: +0.71+0.00j prob: 0.50 Phase:   0.0
```

This shows the correct probability of 0.5 of measuring $|0\rangle$, but the state is still unmodified. Now we *collapse* the state after the measurement, which is more reflective of making an actual, physical measurement:

```
    psi = bell.bell_state(0, 0)
    p0, psi = ops.Measure(psi, 0, 0, collapse=True)
    print('Probability: ', p0)
    psi.dump()
>>
    Probability:  0.49999997
    |00> (|0>):  ampl: +1.00+0.00j prob: 1.00 Phase:   0.0
```

Now only one possible measurement outcome remains, the state $|00\rangle$, which from here on out would be measured with 100% probability.

At this point we have mastered the fundamental concepts and are ready to move on to studying our first quantum algorithms. There are two possible paths forward. You may want to explore Chapter 3 on simple algorithms next before learning more about infrastructure and high-performance simulation in Chapter 4. Or, you may prefer reading Chapter 4 on infrastructure first before exploring Chapter 3 on simple algorithms.

https://doi.org/10.1017/9781009099974.004 Published online by Cambridge University Press