2 Quantum Computing Fundamentals

This chapter outlines the basic principles and rules of quantum computing. In parallel, we develop an initial, easy-to-understand, easy-to-debug code base for building and simulating smaller-scale algorithms.

The chapter is structured as follows. First, we introduce our basic underlying data type, the Python `Tensor` type, which is derived from `numpy`'s `ndarray` data type. Using this type, we construct single qubits and quantum states composed of many qubits. We define operators that allow us to modify states and describe a range of important single-qubit gates. Controlled gates, which play a similar role to 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 computing is expressed in the language of linear algebra, with vectors, matrices, and operations such as the inner product. Quantum algorithms are, to a large degree, algorithms based on linear algebra. Because of that, some of the mathematics is unavoidable. A very compressed summary of necessary mathematical concepts was already presented in the previous chapter. However, since this book has "programmer" in the title, we balance the mathematical development of the algorithms with working code for experimentation.

Let us start by describing a Python data structure that will serve as the basis for all the code in this book. Python may be slow to execute but is fast to develop.[1] It also has the vectorized and accelerated `numpy` numerical library for scientific computing. We make great use of this library and avoid implementing standard numerical linear algebra operations ourselves. In general, we follow Google's coding style guides for Python (Google, 2021b) and C++ (Google, 2021a).

One of the insights here is that it only takes a little bit of code to implement and simulate the algorithms. Of course, there are many existing frameworks and

[1] To be fair, only for programs up to moderate size.

libraries available. The advantage of quickly developing our own framework is that you can focus on learning quantum computing and not be distracted by having to learn another complex framework. Learning quantum computing is already hard enough.

The core data types, such as states and operators, are all vectors and matrices of complex numbers. It is good practice to base the types on one common array abstraction and hide the underlying implementation. The typical benefits are an improved development speed and a smaller jump from experimentation on your laptop to running on a distributed supercomputer. The data type for all subsequent work will be our Python `Tensor` class.

We derive `Tensor` from the `ndarray` array data structure in `numpy`. It will behave just like a `numpy` array, but we can augment it with additional convenience functionality. For example, for ease of debugging, we allow a tensor to have a descriptive name. 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.[2]

Find the code
In file `src/lib/tensor.py`

```
import numpy as np

class Tensor(np.ndarray):
  def __new__(cls, input_array, op_name=None) -> Tensor:
    cls.name = op_name
    return np.asarray(input_array, dtype=tensor_type()).view(cls)

  def __array_finalize__(self, obj) -> None:
    if obj is None:
      return
    # 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 of complex numbers. The choice of which complex data type to use is an interesting question. Should we use complex numbers based on 64-bit doubles, 32-bit floats, or perhaps something else, for example, a TPU 16-bit bfloat[3] format? Smaller data types may be faster to simulate due to lower memory bandwidth requirements, but they come at the cost of reduced numerical precision. The `numpy` package supports `np.complex128` (consisting of two 64-bit doubles) and `np.complex64` (with two 32-bit floats). We define a command line flag that holds the width of the type and functions to return the corresponding `numpy` data type and the number of bits:

[2] Refer to http://numpy.org/doc/stable/user/basics.subclassing.html.
[3] TPU stands for Google's "Tensor Processing Unit," a hardware accelerator for machine learning algorithms. It also introduced the *bfloat* data type, which is a standard fp32 data type but without the lower 16 bits.

```
from absl import flags
flags.DEFINE_integer('tensor_width', 64, 'Width of complex (64, 128)')

def tensor_width():
  return flags.FLAGS.tensor_width

def tensor_type():
  assert tensor_width() == 64 or tensor_width() == 128
  return np.complex64 if tensor_width() == 64 else np.complex128
```

As we shall see in our discussion of quantum states in Section 2.4, the Kronecker product of tensors is an important operation. As mentioned in Section 1.5, this product is commonly referred to as the tensor product, which is also the term we will use.[4] We implement it by adding the member function `kron` to the `Tensor` class. This function delegates to the function of the same name in `numpy`.

We will use this operation in many places, so we additionally overload the Python multiplication operator `*` for convenience. There is the potential to confuse this `*` operator with simple matrix multiplication. However, in Python and in `numpy`, matrix multiplication is done with the *at* operator `@`. We conveniently inherit this multiplication operator from `numpy` and do not have to implement it ourselves:

```
def kron(self, arg: Tensor) -> Tensor:
  return self.__class__(np.kron(self, arg))

def __mul__(self, arg: Tensor) -> Tensor:
  return self.kron(arg)
```

We will often construct larger matrices by tensoring together many *identical* matrices, which corresponds to calling the `kron` function multiple times. To tensor together n matrices A, we will use a notation similar to raising the matrix to the power of n, but add the $\otimes$ operator in the notation:

$$\underbrace{A \otimes A \otimes \cdots \otimes A}_{n} = A^{\otimes n} \quad \neq \quad A^n = \underbrace{AA \cdots A}_{n}.$$

It looks like a power function, but instead of matrix multiplication, it uses Kronecker products. Naming is hard, but this function names itself: We should call it the Kronecker power function, or `kpow` (pronounced "Kah-Pow"). We handle cases where the exponent is 0 as a special case with $x^0 = 1$. As expected, `numpy` correctly computes tensor products with scalars.

```
def kpow(self, n: int) -> Tensor:
  if n == 0:
    return self.__class__(1.0)
  t = self
  for _ in range(n - 1):
```

[4] *Tensoring states* rolls off the tongue much more easily than *Kroneckering states*.

```
    t = np.kron(t, self)
  return self.__class__(t)  # Return a Tensor type
```

Often, especially during testing, we want to compare a `Tensor` with another tensor. We are working with complex numbers based on floating-point data types. Direct comparison of values of these types is considered bad practice due to issues with floating-point precision. Instead, for equality, we have to check that the difference between two numerical values is less than a given ε.

Fortunately, `numpy` comes to the rescue and offers the function `allclose()`, which compares full tensors, so we do not have to iterate over dimensions and compare real and imaginary parts. Here, and in almost all other places, we use a tolerance of 10^{-6} and add the `is_close` method to our `Tensor` type.[5] Python's `math` module has an `isclose()` function. However, we follow Google's coding style, which requires us to name functions with a trailing underscore after `is`, as in `is_close()`:

```
def is_close(self, arg) -> bool:
  return np.allclose(self, arg, atol=1e-6)
```

In Section 1.8, we learned about Hermitian and unitary matrices. The two helper functions below check for these properties:

```
def is_hermitian(self) -> bool:
  if len(self.shape) != 2 or self.shape[0] != self.shape[1]:
    return False
  return self.is_close(np.conj(self.transpose()))

def is_unitary(self) -> bool:
  return Tensor(np.conj(self.transpose()) @ self).is_close(
               Tensor(np.eye(self.shape[0])))
```

Another interesting matrix type is a *permutation matrix*, which has a single 1 in each row and column. Multiplying a column vector by such a matrix allows us to permute the vector elements. The `Tensor` class offers the member function `is_permutation()` to verify this matrix property:

```
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())
```

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

2.2 Qubits

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

First, we must distinguish between a qubit and *state of a qubit*. Physical qubits, developed for real quantum computers, are real physical entities, such as ions captured in an electric field or Josephson junctions in an ASIC. The state of a qubit describes some measurable property of that qubit, such as the energy level of an electron.

In quantum computing, at the level of programming abstractions, the physical implementation does not matter; we are only concerned with the measurable state. This is analogous to classical computing, where very few people care about the quantum effects that enable transistors at the level of logic gates. In this text, we will use the terms qubit and state of the qubit interchangeably.

The state of one or more qubits is often denoted by the Greek symbol $|\psi\rangle$ ("psi"). The standard notation for the 0-state of a qubit is $|0\rangle$ in the Dirac notation and $|1\rangle$ for the 1-state. You can think of these as physically distinguishable states, such as the energy levels of electrons. *Superposition* now means that the state of a qubit is a linear combination of the orthonormal basis[6] states, for example, the $|0\rangle$ and $|1\rangle$ states, as

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

where α and β are complex numbers, called the *probability amplitudes*. We further require that

$$|\alpha|^2 + |\beta|^2 = 1, \tag{2.1}$$

for reasons explained below. Using the basis vectors $(1,\ 0)^T$ and $(0,\ 1)^T$, we define the state of a qubit elegantly as

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

The choice of $(1,\ 0)^T$ and $(0,\ 1)^T$ as orthonormal basis vectors, which is also called the *computational basis*, is intuitive and simplifies many of the calculations. The basis vectors are orthogonal with a scalar product of $\langle 0|1\rangle = 0$ and normalized with scalar products of $\langle 0|0\rangle = \langle 1|1\rangle = 1$.

Other bases are possible, especially those resulting from rotations, which are commonplace in quantum computing. For example, the *Hadamard basis* consists of the two orthonormal vectors $|+\rangle$ and $|-\rangle$, which are defined as

[6] To be rigorous, one would say superposition can be the linear combination of *any* two distinct, not necessarily orthogonal states.

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix},$$

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

For the superposition $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, we required $|\alpha|^2 + |\beta|^2 = 1$. As will become clear later, this follows from one of the fundamental postulates of quantum mechanics, which states that on measurement, the state collapses to $|0\rangle$ with probability $|\alpha|^2$, or to $|1\rangle$ with probability $|\beta|^2$. The state has to collapse to one of the two. The probabilities must add up to a real value of 1. Since amplitudes are complex numbers in general, we use the absolute value of the inner product to calculate the real physical probabilities.

Let us look at a standard example. Suppose we have a qubit in the state

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

For a single complex number c, the conjugate c^* is the same[7] as the Hermitian adjoint $c^\dagger$. The probability $p_{|0\rangle}$ of measuring $|0\rangle$ is the norm squared:[8]

$$p_{|0\rangle} = \left|\frac{\sqrt{3}}{2}\right|^2 = \left(\frac{\sqrt{3}}{2}\right)^\dagger\left(\frac{\sqrt{3}}{2}\right) = \left(\frac{\sqrt{3}}{2}\right)\left(\frac{\sqrt{3}}{2}\right) = \frac{3}{4}.$$

To compute the probability $p_{|1\rangle}$ of measuring $|1\rangle$, we take the norm squared of the amplitude $i/2$ as

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

You can see that the two probabilities of $3/4$ and $1/4$ add up to 1.

The following code will translate these concepts into a straightforward implementation. As a forward reference, we use the type `State`, which we will discuss in Section 2.4. In simple terms, `State` is a vector of complex numbers implemented using `Tensor`.

To construct a qubit, we need α or β, or both. If only one is provided, we can easily compute a candidate[9] for the other one since their squared norms must add up to 1. To compute the squared norms of the complex numbers α and β, we multiply each by its complex conjugate (using `np.conj`). The result will be a real number.[10] To avoid generating a type error from `numpy`, we have to explicitly convert the result to `np.real()`. We compare the results to 1.0, and if it is within tolerance, we construct and return the qubit as a `State`.

[7] We will often use the dagger for simplicity.
[8] We are really computing a projection $|\langle 0|\psi\rangle|^2$, but for more details we will have to wait for Section 2.13.2 on measurements.
[9] Here, we ignore a possible *local phase*, which we will learn about in Section 2.3.
[10] We could also use `np.abs(alpha)**2`, but I prefer it this way; it is more explicit. You will find this construction in many places in this book.

PY **Find the code**
In file `src/lib/state.py`

```
def qubit(alpha: complex = None, beta: complex = None) -> State:
  if alpha is None and beta is None:
    raise ValueError('alpha, beta, or both, need to be specified')
  if beta is None:
    beta = np.sqrt(1.0 - np.real(np.conj(alpha) * alpha))
  if alpha is None:
    alpha = np.sqrt(1.0 - np.real(np.conj(beta) * beta))

  norm2 = np.real(np.conj(alpha) * alpha) + np.real(np.conj(beta) * beta)
  assert math.isclose(norm2, 1.0), 'Qubit probabilities not equal to 1.'
  return State([alpha, beta])
```

2.3 Bloch Sphere

We now introduce the *Bloch sphere*, a 3D visualization of the state of a qubit, named after the famous physicist Felix Bloch, even though it was first introduced by Feynman (1957). It may be especially useful for visual learners. We will use it in the following sections to visualize the effect of operators on qubits. To begin, let us introduce some basic trigonometry and an angle θ. Using

$$\alpha = \cos\frac{\theta}{2} \quad \text{and} \quad \beta = \sin\frac{\theta}{2},$$

we meet the requirement from Equation (2.1) that

$$|\alpha|^2 + |\beta|^2 = \cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2} = 1.$$

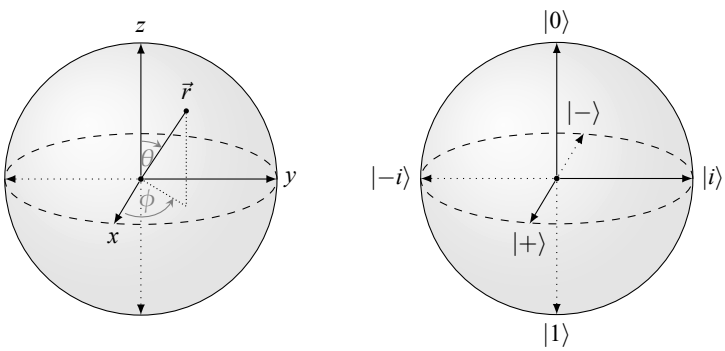
Now we introduce a second angle ϕ as a phase $e^{i\phi}$ between $|0\rangle$ and $|1\rangle$. This phase is called a *local phase* and it plays an important role in many algorithms. We must not ignore it, and, more importantly, Equation (2.1) still holds with it. With this, we can write a qubit in the alternative form

$$|\psi\rangle = e^{i\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle\right). \tag{2.2}$$

The parameters γ and ϕ are real numbers in $[0,2\pi)$ and θ in $[0,\pi)$. The first term $e^{i\gamma}$ in Equation (2.2) is called a *global phase*. Multiplying a state by such a complex coefficient does not have an actual physical meaning because the *expectation value* of the state with or without the coefficient does not change. This is also related to what physicists call *phase invariance*.

The expectation value for an operator A on state $|\psi\rangle$ (which we will develop in Section 2.13 on measurement) is $\langle\psi|A|\psi\rangle$. The Hermitian adjoint of $(c\,|\psi\rangle)^\dagger = \langle\psi|c^*$. We can see that the expectation value with and without a global phase remains unchanged. States with or without a global phase cannot be distinguished:

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



(a) Sphere with axes x, y, and z. (b) Basis states on the sphere.

Figure 2.1 The Bloch sphere representation.

The two parameters θ and ϕ are sufficient to specify a qubit. This leads to a representation as a point on a three-dimensional sphere with unit radius as shown in Figure 2.1(a), where a qubit $|\psi\rangle$ lies on the surface of the sphere. With some trigonometry, we see that this point is specified by a vector $\vec{r} = (\cos\phi \sin\theta, \sin\phi \sin\theta, \cos\theta)$, the so-called *Bloch vector*.

Let us explore where we can find specific states on the sphere, as shown in Figure 2.1(b). The position at the sphere's north pole has $\theta = 0$ and $\phi = 0$ (other values for ϕ will still land the qubit on the north pole, but let us ignore this case for now). With this, the state becomes

$$\cos\frac{0}{2}|0\rangle + e^{i\phi}\sin\frac{0}{2}|1\rangle = 1\,|0\rangle + 0\,|1\rangle = |0\rangle\,.$$

The state $|0\rangle$ sits at the top. Similarly, for $\theta = \pi$, state $|1\rangle$ sits at the south pole:

$$\cos\frac{\pi}{2}|0\rangle + e^{i\phi}\sin\frac{\pi}{2}|1\rangle = 0\,|0\rangle + 1\,|1\rangle = |1\rangle\,.$$

The points where the positive and negative x-axes intersect with the sphere have angles $\theta = \pi/2$ with $\phi = 0$ and $\phi = \pi$. This is where we find the *Hadamard bases* $|+\rangle$ and $|-\rangle$ with

$$\cos\frac{\pi/2}{2}|0\rangle + e^{i\phi}\sin\frac{\pi/2}{2}|1\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = |+\rangle\,,$$

$$\cos\frac{\pi/2}{2}|0\rangle + e^{i\pi}\sin\frac{\pi/2}{2}|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle = |-\rangle\,.$$

Finally, the points that intersect the positive and negative y-axis have angles $\theta = \pi/2$ with $\phi = \pi/2$ and $\phi = 3\pi/2$. These states form another basis which is affectionately, and sometimes confusingly, denoted as $|i\rangle$ and $|-i\rangle$:

$$\cos\frac{\pi/2}{2}|0\rangle + e^{\pi/2}\sin\frac{\pi/2}{2}|1\rangle = \frac{1}{\sqrt{2}}|0\rangle + i\frac{1}{\sqrt{2}}|1\rangle = |\,i\,\rangle \;\; = |{+}y\rangle\,,$$

$$\cos\frac{\pi/2}{2}|0\rangle + e^{3\pi/2}\sin\frac{\pi/2}{2}|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - i\frac{1}{\sqrt{2}}|1\rangle = |{-}i\rangle = |{-}y\rangle\,.$$

The term $|i\rangle$ can be confusing because it is often used to denote arbitrary computational basis states. Instead of $|i\rangle$ and $|-i\rangle$, we will also use the terms $|+y\rangle$ and $|-y\rangle$ for these basis states. To add to the confusion, note that states with anti-parallel Bloch vectors are orthogonal. Orthogonal states do not have orthogonal Bloch vectors.

Another interesting question is how to compute the x,y,z coordinates for a given state $|\psi\rangle$ on a Bloch sphere. We will have to make more progress before we can answer this question in Section 2.7.3. Bloch spheres are only defined for single-qubit states. You can visualize the Bloch sphere of an individual qubit in a multi-qubit system by *tracing out* all the other qubits in the state. This is done with the *partial trace* procedure, a useful tool we introduce in Section 4.3.

2.4 States

As we saw in Section 2.2, the possible *quantum state* of a qubit is a vector of complex numbers that represent probability amplitudes. We should use our trusty `Tensor` class to represent states in code and inherit the `State` class from `Tensor`. In this way, we also conveniently inherit the Python `__repr__` and `__str__` functions from the base class.

Find the code
In file `src/lib/state.py`

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

So far, we have learned how to construct a single-qubit state. But what about a state that consists of multiple qubits? 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 Section 2.1 (implemented as the corresponding Python `__mul__` member function). Given this definition, the quantum state of n qubits is a `Tensor` of 2^n complex probability amplitudes. And we already know, from Equation (1.2), that for two qubits $|\phi\rangle$ and $|\chi\rangle$ we can write the combined state as

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

For two qubits, there are four basis states, and we can write the state $|\psi\rangle$ as[11]

[11] By convention, computational basis states are often denoted as $|e_i\rangle$. Basis states for a general state $|\psi\rangle$ may also be written as $|\psi_0\rangle, \ldots, |\psi_{n-1}\rangle$, which is a convention we will use often.

$$
\begin{aligned}
|\psi\rangle = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} &= c_0 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + c_1 \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + c_2 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + c_3 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \\
&= c_0 |e_0\rangle + c_1 |e_1\rangle + c_2 |e_2\rangle + c_3 |e_3\rangle \\
&= c_0 |\psi_0\rangle + c_1 |\psi_1\rangle + c_2 |\psi_2\rangle + c_3 |\psi_3\rangle \\
&= \sum_{i=0}^{3} c_i |\psi_i\rangle .
\end{aligned}
$$

We already learned in Section 1.5 that the norm of a tensor product of vectors with unit norm is also 1, which is exactly what we need for state vectors to represent probabilities. Probability amplitudes are complex numbers. To compute the inner product, we multiply by the complex conjugates and exploit the fact that the basis states are normalized with an inner product of 1:

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

We can extract an individual value c_i by computing the inner product of the state with the corresponding computational basis vector $|e_i\rangle$. For example, to extract c_2 (you can get c_2^* by reversing the order of the inner product):

$$
\langle e_2|\psi\rangle = \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \end{pmatrix}^T = c_2 . \tag{2.3}
$$

2.4.1 Tensoring States

To build systems of multiple qubits, the individual states of the participating qubits are tensored together. Given our definition of the tensor product in Section 1.3, this was easy to understand when the states were expressed as vectors:

$$
\begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a \begin{pmatrix} c \\ d \end{pmatrix} \\ b \begin{pmatrix} c \\ d \end{pmatrix} \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix} . \tag{2.4}
$$

But what if a state $|\psi\rangle$ is written as an expression, such as

$$
|\psi\rangle = \big(a|0\rangle + b|1\rangle\big) \otimes \big(c|0\rangle + d|1\rangle\big).
$$

Similarly, and sometimes confusingly, the product is often written *without* the operator $\otimes$, as

$$
\big(a|0\rangle + b|1\rangle\big) \otimes \big(c|0\rangle + d|1\rangle\big) \equiv \big(a|0\rangle + b|1\rangle\big)\big(c|0\rangle + d|1\rangle\big).
$$

This can be confusing because it may look like a matrix product or dot product. It is important to be aware of the context. We can "multiply out" the expression, just like a normal product of two terms. As we multiply the two bracketed terms, the scalar factors turn into simple products, and the qubit states are tensored together. For scalar products, the order of their operands does not matter. For qubit states, the ordering must be maintained:

$$\begin{aligned}|\psi\rangle &= \left(a\,|0\rangle + b\,|1\rangle\right)\left(c\,|0\rangle + d\,|1\rangle\right)\\ &= a\,|0\rangle\left(c\,|0\rangle + d\,|1\rangle\right) + b\,|1\rangle\left(c\,|0\rangle + d\,|1\rangle\right)\\ &= ac\,|00\rangle + ad\,|01\rangle + bc\,|10\rangle + bd\,|11\rangle\,.\end{aligned}$$

Writing this state as a vector results in the same state vector $\begin{pmatrix} ac & ad & bc & bd \end{pmatrix}^T$ as in Equation (2.4). To make the required ordering clear, individual qubits sometimes get a subscript, indicating who they belong to, such as Alice or Bob:[12]

$$\begin{aligned}|\psi\rangle &= \left(a\,|0_A\rangle + b\,|1_A\rangle\right)\left(c\,|0_B\rangle + d\,|1_B\rangle\right)\\ &= ac\,|0_A0_B\rangle + ad\,|0_A1_B\rangle + bc\,|1_A0_B\rangle + bd\,|1_A1_B\rangle\,.\end{aligned}$$

The multiplication procedure can be reversed; we can *factor out* individual qubits. This should not be surprising, but it may be helpful to see it at least once:

$$\begin{aligned}|\psi\rangle &= ac\,|00\rangle + ad\,|01\rangle + bc\,|10\rangle + bd\,|11\rangle\\ &= |0\rangle\left(ac\,|0\rangle + ad\,|1\rangle\right) + |1\rangle\left(bc\,|0\rangle + bd\,|1\rangle\right)\\ &= a\,|0\rangle\left(c\,|0\rangle + d\,|1\rangle\right) + b\,|1\rangle\left(c\,|0\rangle + d\,|1\rangle\right)\\ &= \left(a\,|0\rangle + b\,|1\rangle\right)\left(c\,|0\rangle + d\,|1\rangle\right).\end{aligned}$$

You will find these types of state manipulations in several places in this book.

2.4.2 Qubit Ordering

As we compose states of multiple qubits, we must consider the issue of *endianness*. Consider how the bits in a typical byte are numbered, with the least significant bit 0 on the right, as shown in Figure 2.2(a). However, when we think of arrays, we typically have element 0 at address 0 at the top of the array, as shown in Figure 2.2(b).

Classical binary numbers are combinations 0s or 1s, which we interpret as an n-ary number. In quantum computing, we can also combine multiple qubits in the basis states $|0\rangle$ and $|1\rangle$ with the tensor product and interpret the resulting state as a classical binary number. There are two distinct conventions:

1. In the *little-endian* convention, the least significant part of a data structure is placed at the lowest address. The Intel x86 family of CPUs follows this convention. The hexadecimal value `0x1234` is stored in a 16-bit memory space as `0x3412`, with the least significant byte `0x34` at the lower byte address.

[12] Alice and Bob are widely used as stand-ins to denote two distinct systems A and B.