3 Simple Algorithms

In this section, we introduce a first set of quantum algorithms. All that is needed to follow this section is the background and infrastructure from Chapter 2. What makes the algorithms presented here *simple* compared to the *complex* algorithms in Chapter 6? A judgment call. To justify the judgement call, the algorithms in this chapter are typically shorter and require less preparation or background than those in later chapters. Additionally, the derivations are developed with great detail. Many of these techniques will be taken for granted in later sections.

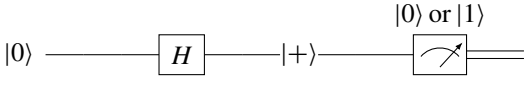
In this chapter, we start with what is possibly the most simple algorithm of all: a quantum random number generator. We follow this with a range of gate equivalences – how one gate, or gate sequence, can be expressed by another. Armed with these basic tools, we implement a classical full adder but with quantum gates. This circuit does not yet exploit superposition or entanglement.

Then it gets more exciting. We describe a swap test, which allows measurement of the similarity between two states without directly measuring the states themselves. We describe two algorithms with very cool names that utilize entanglement – quantum teleportation and superdense coding.

After this we move on to three so-called oracle algorithms. These algorithms exploit superposition and compute solutions in parallel using a large unitary operator. These are the first quantum algorithms we explore that perform better than their classical counterparts.

3.1 Random Number Generator

Every programming system introduces itself with the equivalent of a "Hello World" program. In quantum computing, this may be a random number generator. It is the simplest possible quantum circuit that does something meaningful, and it does so with just one qubit and one gate:



https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

The Hadamard gate will put the state into superposition:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle. \tag{3.1}$$

On measurement,[1] the state will collapse to either $|0\rangle$ or $|1\rangle$ with exactly 50% probability for each case. Remember that after applying the Hadamard gate, the probability amplitudes for each of the two possible resulting states are $1/\sqrt{2} = 0.707\ldots$. Recall that the probability is the square of the norm: for amplitude a, $p = |a|^2 = a^*a$. You can validate this with a simple code experiment:

```
  psi = ops.Hadamard()(state.zero)
  psi.dump()
>>
0.70710678+0.00000000i  |0>  50.0%
0.70710678+0.00000000i  |1>  50.0%
```

Since we can construct one single, completely random qubit, which we interpret as a classical bit after measurement, bundling multiple of these bits in parallel or sequence allows the generation of random numbers of any bit width. By *random*, we mean true, atomic-level randomness, not classical pseudo-randomness. There is a finite number of possible (enumerable) states in classical computers with a finite amount of memory. Hence all random numbers generated on a classical computer are not *truly* random; their sequence will repeat itself eventually.

This circuit can barely be called a circuit, never mind an algorithm (even though in Section 6.8 on amplitude amplification, we do call it an algorithm). It only has one gate, so it is the simplest of all possible circuits. Nevertheless, it exploits crucial quantum computing properties, namely superposition and the probabilistic collapse of the wave function on measurement. It is trivial, and it is not, both at the same time. It is a true quantum circuit.

3.2 Gate Equivalences

As we learned earlier, points on the surface of the Bloch sphere can be reached in an infinite number of ways, simply by utilizing rotations. Similarly for circuits, using only standard gates, there are many interesting equivalences for one-, two-, and many-more-qubit circuits. In this section, we describe common equivalences, mostly in the form of code. With just a little stretch of the imagination, we can consider those simple circuits as algorithms; that's why we're discussing them in this part of the book.

There are also higher-level functions that can be composed of simpler, one- or two-qubit gates, e.g., a Swap gate over larger qubit distances and the double-controlled X-gate. This aspect, the reduction to one- and two-qubit gates, is important, and in

[1] In the Z-basis – we talk more about this in later sections.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

the remainder of this text, we will (mostly) focus on these types of gates. The main reasons for this focus are:

- Two-qubit gates are already tough to implement on a physical quantum computer. Larger ones even more so.
- One key result in quantum computing proves that any unitary gate can be approximated up to arbitrary accuracy by single- and double-qubit gates only.
- We want peak simulation performance, which can be achieved with these types of gates.

The example code in the following sections can be quite trivial. We still show it, it should encourage you to run your own experiments!

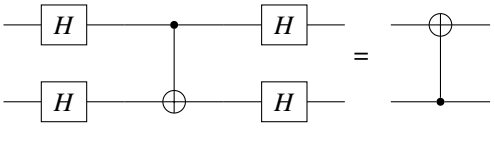
3.2.1 Root of Gate Squared = Gate

Squaring the root of a gate should result in the gate itself. The root of a rotation is a rotation by half the angle. We can check that $T^2 = S, S^2 = Z$, and $T^4 = Z$. Note that the adjoint of a rotation is a rotation in the other direction. In the code below, we also validate the root of the X-gate as $V^2 = X$:

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

  s = ops.Tgate() @ ops.Tgate()
  self.assertTrue(s.is_close(ops.Phase()))

def test_s_gate(self):
  """S^2 == Z."""

  x = ops.Sgate() @ ops.Sgate()
  self.assertTrue(s.is_close(ops.PauliZ()))

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

  s = ops.Vgate() @ ops.Vgate()
  self.assertTrue(s.is_close(ops.PauliX()))
```

3.2.2 Inverted Controlled-Not

We can turn a $CNOT_{a,b}$ into a $CNOT_{b,a}$ by applying a Hadamard gate to the left and right of the $CNOT$ on both qubits:



https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

```
def test_had_cnot_had(self):
    h2 = ops.Hadamard(2)
    cnot = ops.Cnot(0, 1)
    op = h2(cnot(h2))
    self.assertTrue(op.is_close(ops.Cnot(1, 0)))
```

We can also convince ourselves of this result by looking at the operator matrices; this is a useful debugging technique. The circuit above translates to the following gate-level expression. Because this circuit is symmetric around the *CNOT* gate, we don't have to worry about the ordering of the matrix multiplications:

$$(H \otimes H)\ CNOT_{0,1}\ (H \otimes H).$$

In matrix form:

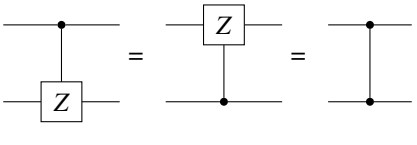
$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

We construct this in code and find that both versions produce the same operator matrix:

```
    (ops.Hadamard(2) @ ops.Cnot(0, 1) @ ops.Hadamard(2)).dump()
>>
 1.0       -         -         -
 -         -         -         1.0
 -         -         1.0       -
 -         1.0       -         -

    ops.Cnot(1, 0).dump()
>>
 1.0       -         -         -
 -         -         -         1.0
 -         -         1.0       -
 -         1.0       -         -
```

3.2.3 Controlled-Z Gate

The controller and controlled qubits for a Controlled-Z gate can be swapped without affecting the results. Therefore, this controlled gate is often drawn without direction using just two black dots:



https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

Let us confirm that the Controlled-Z is indeed symmetric. In Section 2.7 on controlled gates we learned how to construct controlled unitary gates with the help of projectors such as:

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

It is a good exercise to try and manually compute the tensor products for this Controlled-Z gate. In one case we add the matrices:

$$CZ_{0,1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

In the case with the indices changed from $0,1$ to $1,0$ we add these matrices:

$$CZ_{1,0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

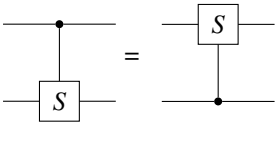
In both cases the result will be this operator matrix:

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

The corresponding code for this experiment is below. Note that constructing a *Multi-Controlled-Z* results in a similar matrix with all diagonal elements being 1 except the bottom-right element, which is -1. Try it out yourself!

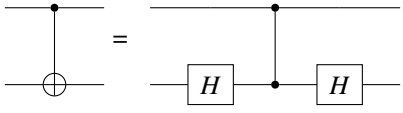
```
def test_controlled_z(self):
  z0 = ops.ControlledU(0, 1, ops.PauliZ())
  z1 = ops.ControlledU(1, 0, ops.PauliZ())
  self.assertTrue(z0.is_close(z1))
```

Note that all controlled phase gates are symmetric, for example:



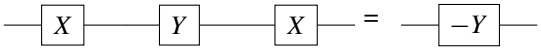
A related gate equivalence is the following, which allows the construction of Controlled-Not gates using Hadamard gates and Controlled-Z gates:

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press



3.2.4 Negate Y-Gate

We can *negate* the Y-gate with help of two X-gates to its left and right:




```
def test_xyx(self):
    x = ops.PauliX()
    y = ops.PauliY()
    print(y)

    op = x(y(x))
    print(op)
    self.assertTrue(op.is_close(-1.0 * y))
```

The two print statements in the code above indeed produce the expected result:

```
Operator for 1-qubit state space. Tensor:
[[ 0.+0.j -0.-1.j]
 [ 0.+1.j  0.+0.j]]
Operator for 1-qubit state space. Tensor:
[[0.+0.j 0.+1.j]
 [0.-1.j 0.+0.j]]
```

3.2.5 Pauli Matrix Relations

Placing Hadamard operators to the left and right of a Pauli matrix yields another Pauli matrix in the following ways. In general, writing operators next to each other means they follow function call syntax. So the mathematical ABC would be written as `A(B(C))`. The expressions below are symmetric, so the order does not matter in this instance.[2]

$$HXH = Z,$$
$$HYH = -Y = XYX,$$
$$HZH = X.$$

[2] Note that these equivalences are related to the *Pauli commutators*. See, for example, https://en.wikipedia.org/wiki/Pauli_matrices#Commutation_relations.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

A particular equality that we will encounter in several places is the following:

$$XZ = iY. \tag{3.2}$$

You can validate these results with a short code snippet:

```
  def test_equalities(self):
    # Generate the Pauli and Hadamard matrices.
    _, x, y, z = ops.Pauli()
    h = ops.Hadamard()

    # Check equalities.
    op = h(x(h))
    self.assertTrue(op.is_close(z))

    op = h(y(h))
    self.assertTrue(op.is_close(-1.0 * y))

    op = h(z(h))
    self.assertTrue(op.is_close(x))

    op = x(z)
    self.assertTrue(op.is_close(1.0j * y))
```

3.2.6 Change Rotation Axis

The following equivalence is interesting because it shows the effects of a global phase on an operator. A T-gate represents a rotation about the z-axis by $\pi/4$. Bracketing the T-gate with Hadamard gates turns the combined unitary gate into a rotation about the x-axis by $\pi/4$:

$$HTH = R_x(\pi/4).$$

The combined HTH is this operator:

```
Operator for 1-qubit state space. Tensor:
[[0.8535533 +0.35355335j 0.14644662-0.35355335j]
 [0.14644662-0.35355335j 0.8535533 +0.35355335j]]
```

Let us compare this against a rotation by $\pi/4$ around the x-axis, which is this operator:

```
Operator for 1-qubit state space. Tensor:
[[0.9238795+0.j         0.       -0.38268343j]
 [0.       -0.38268343j 0.9238795+0.j        ]]
```

At first glance, it does not appear that these two results are equivalent. However, dividing one by the other results in a matrix with all equal elements. This means that the two operators only differ by a multiplication factor. Applying a scaled operator to a

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

state leads to an equally scaled state. Since this is a global phase, which has no physical meaning, we can ignore it. Conversely, we can say that operators that differ only by a constant factor are equivalent. If this is true then computing the division between the two should yield a matrix with all identical elements. And indeed, it does:

```
def test_global_phase(self):
    h = ops.Hadamard()
    op = h(ops.Tgate()(h))

    # If equal up to a global phase, all values should be equal.
    phase = op / ops.RotationX(math.pi/4)
    self.assertTrue(math.isclose(phase[0, 0].real, phase[0, 1].real,
                                 abs_tol=1e-6))
    self.assertTrue(math.isclose(phase[0, 0].imag, phase[0, 1].imag,
                                 abs_tol=1e-6))
    self.assertTrue(math.isclose(phase[0, 0].real, phase[1, 0].real,
                                 abs_tol=1e-6))
    self.assertTrue(math.isclose(phase[0, 0].imag, phase[1, 0].imag,
                                 abs_tol=1e-6))
    self.assertTrue(math.isclose(phase[0, 0].real, phase[1, 1].real,
                                 abs_tol=1e-6))
    self.assertTrue(math.isclose(phase[0, 0].imag, phase[1, 1].imag,
                                 abs_tol=1e-6))
```

3.2.7 Controlled-Controlled Gates

As long as a root $R = \sqrt{U}$ exists for a unitary gate U (it always does[3]), we can construct a double-controlled U by using two-qubit gates only. This is important for simulation performance, because two-qubit gates can be simulated very efficiently. Furthermore, for physical machines, building gates with more than two qubits is a major challenge, if not impossible.

An example of a double-controlled gate is the Toffoli gate from Section 2.7.2. It is a double-controlled X-gate, as shown on the left-hand side of Figure 3.1. It can be built with the so-called Sleator–Weinfurter construction (Barenco et al., 1995), which is illustrated on the right-hand side of Figure 3.1. Note that the circuit only uses two-qubit gates!

We know that the square root of the X-gate is the V-gate, and we also have the `adjoint()` function to compute the adjoint of this gate, or any other tensor. This construction works for any single-qubit gate and its root, so we can construct double-controlled X, Y, Z, T, and any other controlled 2×2 gates. Note that for now we are using the fixed indices 0, 1, 2 for the qubits in the code below. A general construction will follow later.

[3] We can prove that *any* unitary matrix has a square root. Unitary matrices are diagonalizable, so the root of the unitary matrix is just the root of the diagonal elements.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

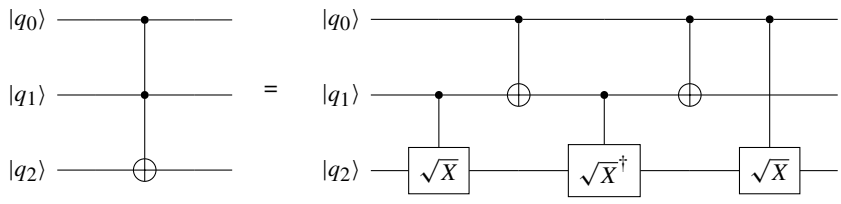


Figure 3.1 The Sleator–Weinfurter construction for a double-controlled X-gate.

```
def test_v_vdag_v(self):
  # Make Toffoli out of V = sqrt(X).
  #
  v = ops.Vgate()  # Could be any unitary, in principle!
  ident = ops.Identity()
  cnot = ops.Cnot(0, 1)

  o0 = ident * ops.ControlledU(1, 2, v)
  c2 = cnot * ident
  o2 = (ident * ops.ControlledU(1, 2, v.adjoint()))
  o4 = ops.ControlledU(0, 2, v)
  final = o4 @ c2 @ o2 @ c2 @ o0

  v2 = v @ v
  cv1 = ops.ControlledU(1, 2, v2)
  cv0 = ops.ControlledU(0, 1, cv1)
  self.assertTrue(final.is_close(cv0))
```

3.2.8 Multi-Controlled Gates

We already mentioned (Section 3.2.7) that the ability to construct double-controlled gates with only two-qubit gates is important. The logical next step is to construct multi-controlled gates.

How to go about this? There is an elegant construction for an n-way controlled gate that requires $n - 2$ ancilla qubits, all initialized to $|0\rangle$. Let's look at the first half of the circuit in Figure 3.2.

The cascading Toffoli gates build up the final predicate in the top-most ancilla qubit to control the X-gate at the bottom. Only if all predicate qubits were $|1\rangle$ will the top-most qubit be $|1\rangle$ as well.

This construction can be used to control other single-qubit gates. A potential problem is that the system's state is now entangled with the ancilla qubits. A solution for this problem, which is detailed in Section 2.13, is to *uncompute* the cascade of Toffoli gates by computing the gates' adjoints and applying them in reverse order. By doing this, as shown in the right half of Figure 3.2, the ancilla qubits are being reversed to their initial state. The state can again be expressed as a product state, and all

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

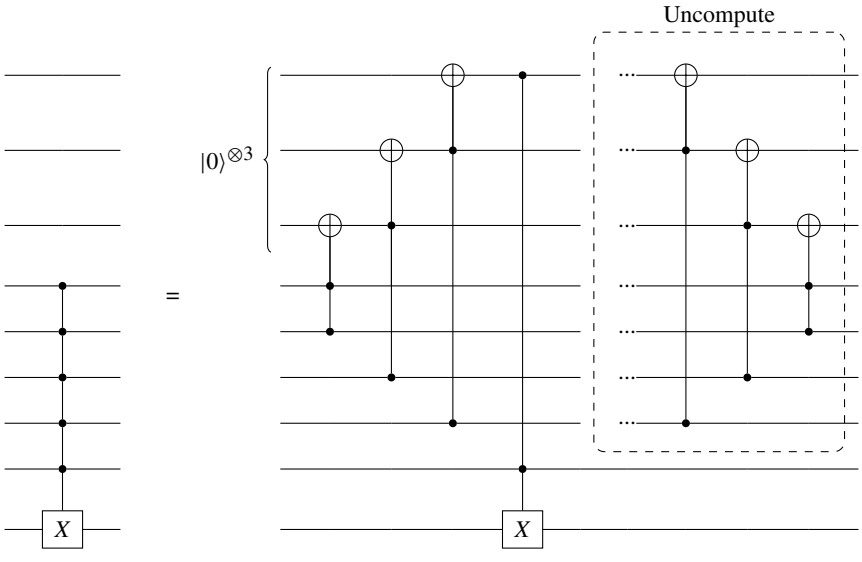


Figure 3.2 A multi-controlled X-gate.

entanglement with the ancilla qubits has been eliminated. We detail an implementation of multi-controlled gates that may have 0, 1, or many controllers and that can be controlled by $|0\rangle$ or $|1\rangle$, in Section 4.3.7.

Other constructions are possible. Mermin (2007) proposes multi-controlled gates that trade additional gates for lower numbers of ancillae, as well as circuits that don't require the ancillae to be in $|0\rangle$ states (which may save a few uncomputation gates).

3.2.9 Equivalences of Controlled Gates

In this section we list several equalities for controlled gates, using this shorthand notation:

- C_x is a Controlled-Not from qubit 0 to qubit 1.
- X_0 is the X-gate applied to qubit 0. Similar for index 1, and the Y-gate and Z-gate.

We again write ABC as a short form of $A(B(C))$[4]:

$$
\begin{aligned}
C_x X_0 C_x &= X_0 X_1, \\
C_x Y_0 C_x &= Y_0 X_1, \\
C_x Z_0 C_x &= Z_0, \\
C_x X_1 C_x &= X_1, \\
C_x Y_1 C_x &= Z_0 Y_1, \\
C_x Z_1 C_x &= Z_0 Z_1,
\end{aligned}
$$

[4] But note that matrix multiplication is associative.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

$$R_{z,0}(\phi)C_x = C_x R_{z,0}(\phi),$$
$$R_{x,1}(\phi)C_x = C_x R_{x,1}(\phi).$$

We can use the following code to validate these equivalences. We will also revisit some of these in Section 8.4 where we discuss compiler optimization.

```
def test_control_equalities(self):
    """Exercise 4.31 Nielson, Chuang."""

    i, x, y, z = ops.Pauli()
    x1 = x * i
    x2 = i * x
    y1 = y * i
    y2 = i * y
    z1 = z * i
    z2 = i * z
    c = ops.Cnot(0, 1)
    theta = 25.0 * math.pi / 180.0
    rx2 = i * ops.RotationX(theta)
    rz1 = ops.RotationZ(theta) * i

    self.assertTrue(c(x1(c)).is_close(x1(x2)))
    self.assertTrue((c @ x1 @ c).is_close(x1 @ x2))
    self.assertTrue((c @ y1 @ c).is_close(y1 @ x2))
    self.assertTrue((c @ z1 @ c).is_close(z1))
    self.assertTrue((c @ x2 @ c).is_close(x2))
    self.assertTrue((c @ y2 @ c).is_close(z1 @ y2))
    self.assertTrue((c @ z2 @ c).is_close(z1 @ z2))
    self.assertTrue((rz1 @ c).is_close(c @ rz1))
    self.assertTrue((rx2 @ c).is_close(c @ rx2))
```

3.2.10 Swap Gate

For completeness, let's go over the Swap gate one more time. It is constructed with three Controlled-Not gates in series, as shown in Figure 3.3.

```
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)
```

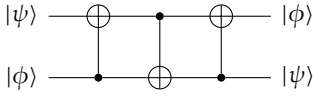


Figure 3.3 Construction of a swap gate with three Controlled-Not gates.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

There are many more equivalences to be found in the literature. They are all interesting and valuable, especially in the context of optimization and compilation. An interesting (and not necessarily trivial) problem is to find them programmatically. We leave this as a challenge to you.

3.3 Classical Arithmetic

In this section we study a standard classical logic circuit, the full adder, and implement it with quantum gates. The quantum circuit does not exploit any of the quantum features – we will detail arithmetic in the quantum Fourier domain in Section 6.2. A 1-bit full adder block is usually drawn as shown in Figure 3.4.

Input bits are A and B. The carry-in from a previous, chained-in full adder is denoted by C_{in}. The outputs are the sum Sum and the potential carry-out C_{out}. Multiple full adders can be chained together (with C_{in} and C_{out}) to produce adders of arbitrary bit width. The truth table for the full adder logic circuit is shown in Table 3.1.

Classical circuits use, unsurprisingly, classical gates like AND, OR, NAND, and others. The task at hand is to construct a quantum circuit that produces the same truth table by using only quantum gates. Classical 0s and 1s are represented by the basis states $|0\rangle$ and $|1\rangle$. With some thought (and experimentation), we arrive at the circuit in Figure 3.5. Note that this circuit exploits neither superposition nor entanglement.

Table 3.1 Truth table for the full adder logic circuit.

A	B	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

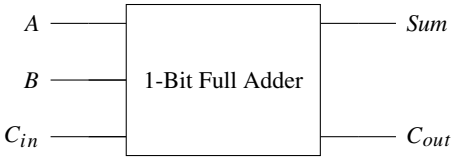


Figure 3.4 The 1-bit full adder block diagram.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

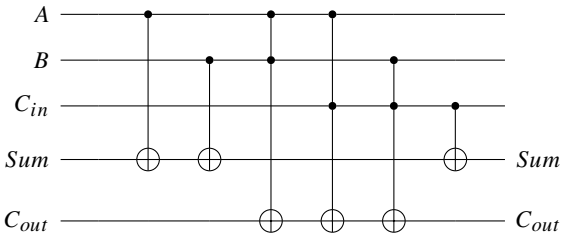


Figure 3.5 Classical full adder, implemented with quantum gates.

Let's walk through the circuit to convince ourselves that it is working properly:

- If A is 1, *Sum* will be flipped to 1 (*CNOT* from A to *Sum*).
- If B is 1, *Sum* will be flipped to 1 or flipped back to 0 if it was set to 1 already.
- If C_{in} is 1, *Sum* will be flipped one more time with the *CNOT* at the very right.
- C_{out} will be flipped if both A and B are set, or both A and C_{in} are set, or both B and C_{in} are set.
- What happens if all A, B, and C_{in} are set? *Sum* will start as 0 and go through these states: $0, 1, 0, 1$. C_{out} will also start as 0 and go through these states: $0, 1, 0, 1$. The final result is 1 and 1 for the two signals.

The implementation is straightforward with Controlled-Not gates and double-controlled X-gates. Measurements are probabilistic, but in this case the probability for the correct result is 100%. There is only a single state with nonzero probability.

Let us use our infrastructure to implement this circuit (the sources are in file `src/arith_classic.py` in the open-source repository). We apply each gate to the state in the order shown in Figure 3.5:

```
def fulladder_matrix(psi: state.State):
  """Non-quantum-exploiting, classic full adder."""

  psi = ops.Cnot(0, 3)(psi, 0)
  psi = ops.Cnot(1, 3)(psi, 1)
  psi = ops.ControlledU(0, 1, ops.Cnot(1, 4))(psi, 0)
  psi = ops.ControlledU(0, 2, ops.Cnot(2, 4))(psi, 0)
  psi = ops.ControlledU(1, 2, ops.Cnot(2, 4))(psi, 1)
  psi = ops.Cnot(2, 3)(psi, 2)
  return psi
```

We run an experiment the following way. First, we construct the state from the inputs, augmenting it with two $|0\rangle$ states for expected outputs `sum` and `cout`. Then, we apply the circuit we just constructed. We measure the probabilities of the outputs being 1, which means we will get a probability of 0.0 if the state was $|0\rangle$ and a probability of 1.0 if the state was $|1\rangle$:

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

```
def experiment_matrix(a: int, b: int, cin: int,
                      expected_sum: int, expected_cout: int):
  """Run a simple classic experiment, check results."""

  psi = state.bitstring(a, b, cin, 0, 0)
  psi = fulladder_classic(psi)

  bsum, _ = ops.Measure(psi, 3, tostate=1, collapse=False)
  bout, _ = ops.Measure(psi, 4, tostate=1, collapse=False)
  print(f'a: {a} b: {b} cin: {cin} sum: {bsum} cout: {bout}')
  if bsum != expected_sum or bout != expected_cout:
    raise AssertionError('invalid results')
```

We check the circuit for all inputs:

```
def add_classic():
  """Full eval of the full adder."""

  for exp_function in [experiment_matrix]:
    exp_function(0, 0, 0, 0, 0)
    exp_function(0, 1, 0, 1, 0)
    exp_function(1, 0, 0, 1, 0)
    exp_function(1, 1, 0, 0, 1)
    [...]

def main(argv):
  [...]
  add_classic()
```

This will produce the following output. The absence of error messages indicates that things went as planned:

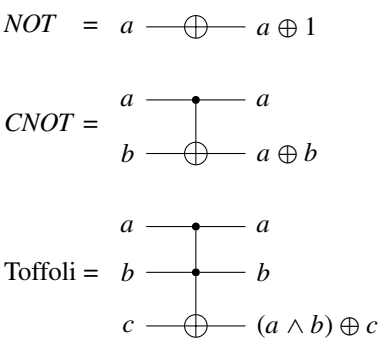
```
a: 0 b: 0 cin: 0 sum: 0.0 cout: 0.0
a: 0 b: 1 cin: 0 sum: 1.0 cout: 0.0
a: 1 b: 0 cin: 0 sum: 1.0 cout: 0.0
a: 1 b: 1 cin: 0 sum: 0.0 cout: 1.0
[...]
```

Other classical circuits can be implemented and combined in this way to build up more powerful circuits. We show a general construction below. All these circuits point to a general statement about quantum computers: since *universal* logic gates can be implemented on quantum computers, a quantum computer is as least as capable as a classical computer. It does not mean that it performs better in the general case. The circuit presented here may be a very inefficient way to implement a simple 1-bit adder. However, we will soon see a class of algorithms that performs significantly better on quantum computers than on classical computers.

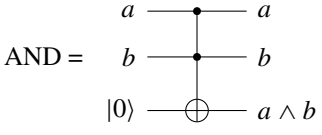
https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

3.3.1 General Construction of Logic Circuits

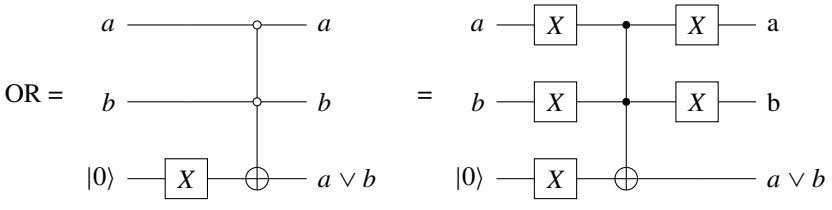
Here is a general construction to express classical logic circuits with quantum gates (Williams, 2011). This method uses a small set of only three quantum gates:



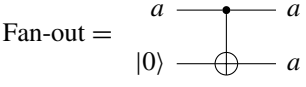
These three gates are sufficient to construct the classical gates AND ($\wedge$), OR ($\vee$), and, of course, the NOT gate. The AND gate is a Toffoli gate with a $|0\rangle$ as third input:



The OR gate is slightly more involved, but it is just based on another Toffoli gate:



We know that with NOT and AND, we can build the universal NAND gate, which means we can construct any classical logic circuit with quantum gates. We might run into the need for fan-out to connect single wires to multiple gates for complex logic circuits. So, we need a *fan-out* circuit. Is this even possible? Doesn't fan-out violate the no-cloning theorem? The answer is no, it does not, because logical 0 and 1 are represented by qubits in states $|0\rangle$ or $|1\rangle$. For those basis states, cloning and fan-out are indeed possible, as we showed in Section 2.13.



With these elements and knowing that any Boolean formula can be expressed as a product of sums,[5] we can build any logic circuit with quantum gates. Of course,

[5] https://en.wikipedia.org/wiki/Canonical_normal_form

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

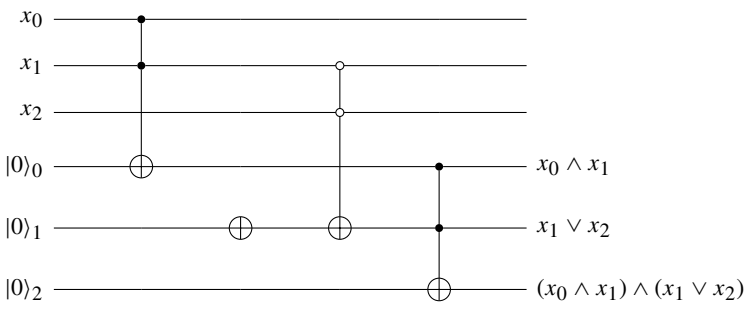


Figure 3.6 A Boolean formula, expressed with quantum gates.

making this construction efficient would require additional techniques, such as ancilla management, uncomputation, and general minimization of gates.

An example of a quantum circuit for the Boolean formula $(x_0 \wedge x_1) \wedge (x_1 \vee x_2)$ is shown in Figure 3.6. In this circuit diagram, we do not show the uncomputation following the final gate that would be required to disentangle the ancillae from the state. The ability to uncompute ancillae in a large chain of logic expressions can reduce the number of required ancillae. In this example, we could uncompute $|0\rangle_0$ and $|0\rangle_1$ and make them available again for future temporary results.

3.4 Swap Test

The quantum swap test allows measuring similarity between two quantum states without actually measuring the two states directly (Buhrman et al., 2001). If the resulting measurement probability is close to 0.5 for the basis state $|0\rangle$ of an *ancilla*, it means that the two states are very different. Conversely, a measurement probability close to 1.0 means that the two states are very similar. In the physical world, we have to run the experiment multiple times to confirm the probabilities. In our implementation, we can conveniently peek at the probabilities.

This is an instance of a quantum algorithm that allows the derivation of an *indirect* measure. It won't tell us what the two states are – that would constitute a measurement. It also does not tell us which state has the larger amplitude. However, it does tell us how similar two unknown states are without having to measure them. The circuit to measure the proximity of qubits $|\psi\rangle$ and $|\phi\rangle$ is shown in Figure 3.7.

Let's denote the state of the 3-qubit system by χ and see how it progresses going from left to right. We use this circuit as a first example to exhaustively derive the related math.

At the start of the circuit, the state is the tensor product of the three qubits:

$$|\chi_0\rangle = |0, \psi, \phi\rangle.$$

The Hadamard gate on qubit 0 superimposes the system to:

$$|\chi_1\rangle = \frac{1}{\sqrt{2}}\big(|0, \psi, \phi\rangle + |1, \psi, \phi\rangle\big).$$

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

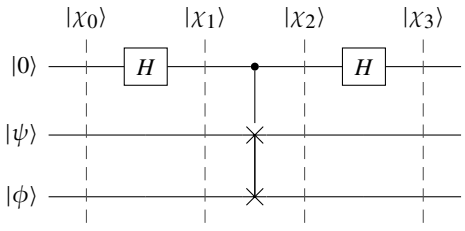


Figure 3.7 The swap test circuit.

The Controlled-Swap gate modifies the second half of this expression because of the controlling $|1\rangle$ state for qubit 0. In the part marked b, $|\phi\rangle$ and $|\psi\rangle$ switch positions:

$$|\chi_2\rangle = \frac{1}{\sqrt{2}}\Big(\underbrace{|0,\psi,\phi\rangle}_{a} + \underbrace{|1,\phi,\psi\rangle}_{b}\Big).$$

The second Hadamard now superimposes further. The first part of the state (marked as a),

$$\frac{1}{\sqrt{2}}\Big(\underbrace{|0,\psi,\phi\rangle}_{a} + \cdots,$$

turns into the following (the Hadamard superposition of the $|0\rangle$ state introduces a plus sign)

$$\frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\big(|0,\psi,\phi\rangle + |1,\psi,\phi\rangle\big) + \cdots.$$

The second part (marked as b):

$$\cdots + \underbrace{|1,\phi,\psi\rangle}_{b}\Big),$$

becomes the following state (the Hadamard superposition of $|1\rangle$ introduces a minus sign)

$$\cdots + \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\big(|0,\phi,\psi\rangle - |1,\phi,\psi\rangle\big).$$

Combining the two sub expressions results in state $|\chi_3\rangle$:

$$|\chi_3\rangle = \frac{1}{2}\big(|0,\psi,\phi\rangle + |1,\psi,\phi\rangle + |0,\phi,\psi\rangle - |1,\phi,\psi\rangle\big).$$

This simplifies to the following, pulling out the first qubit (qubit 0):

$$|\chi_3\rangle = \frac{1}{2}|0\rangle\big(|\psi,\phi\rangle + |\phi,\psi\rangle\big) + \frac{1}{2}|1\rangle\big(|\psi,\phi\rangle - |\phi,\psi\rangle\big).$$

Now we measure the first qubit. If it collapses to $|0\rangle$, the second term disappears (it can no longer be measured and has probability 0). We *only* consider measurements

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

that result in state $|0\rangle$ for the first qubit and ignore all others. The probability amplitude of the state collapsing to state $|0\rangle$ is taken from the first term:

$$\frac{1}{2}|0\rangle\big(|\psi,\phi\rangle + |\phi,\psi\rangle\big).$$

The probability is computed via squaring the amplitude's norm, which is to multiply the amplitude with its complex conjugate. Here we have to be careful: to compute this square (of amplitude and its complex conjugate) we have to square the whole amplitude to the $|0\rangle$ state, which includes the two tensor products after the $|0\rangle$ itself, as well as the factor of $1/2$:

$$\begin{aligned}
&\frac{1}{2}\big(|\psi,\phi\rangle + |\phi,\psi\rangle\big)^\dagger \frac{1}{2}\big(|\psi,\phi\rangle + |\phi,\psi\rangle\big) \\
&\quad = \frac{1}{2}\big(\langle\psi,\phi| + \langle\phi,\psi|\big)\frac{1}{2}\big(|\phi,\psi\rangle + |\psi,\phi\rangle\big) \\
&\quad = \frac{1}{4}\langle\psi,\phi|\phi,\psi\rangle + \frac{1}{4}\underbrace{\langle\psi,\phi|\psi,\phi\rangle}_{=1} + \frac{1}{4}\underbrace{\langle\phi,\psi|\phi,\psi\rangle}_{=1} + \frac{1}{4}\langle\phi,\psi|\psi,\phi\rangle.
\end{aligned}$$

The scalar product of a normalized state with itself is 1.0, which means, in the above expression, that the second and third subterms each become $\frac{1}{4}$ and the expression simplifies to:

$$\frac{1}{2} + \frac{1}{4}\langle\psi,\phi|\phi,\psi\rangle + \frac{1}{4}\langle\phi,\psi|\psi,\phi\rangle. \tag{3.3}$$

Now recall how to compute the inner product of two tensors from Equation (1.7). Given two states:

$$\begin{aligned}
|\psi_1\rangle &= |\phi_1\rangle \otimes |\chi_1\rangle, \\
|\psi_2\rangle &= |\phi_2\rangle \otimes |\chi_2\rangle,
\end{aligned}$$

we compute the inner product as:

$$\langle\psi_1|\psi_2\rangle = \langle\phi_1|\phi_2\rangle\langle\chi_1|\chi_2\rangle.$$

This means we rewrite Equation (3.3) as the following, changing the order of the scalar products; they are just complex numbers:

$$\begin{aligned}
&\frac{1}{2} + \frac{1}{4}\langle\psi,\phi|\phi,\psi\rangle + \frac{1}{4}\langle\phi,\psi|\psi,\phi\rangle \\
&\quad = \frac{1}{2} + \frac{1}{4}\langle\psi|\phi\rangle\langle\phi|\psi\rangle + \frac{1}{4}\langle\phi|\psi\rangle\langle\psi|\phi\rangle \\
&\quad = \frac{1}{2} + \frac{1}{4}\langle\psi|\phi\rangle\langle\phi|\psi\rangle + \frac{1}{4}\langle\psi|\phi\rangle\langle\phi|\psi\rangle \\
&\quad = \frac{1}{2} + \frac{1}{2}\langle\psi|\phi\rangle\langle\phi|\psi\rangle.
\end{aligned}$$

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

Now let us think about the square of an inner product:

$$\langle\psi|\phi\rangle^2 = \langle\psi|\phi\rangle^*\langle\psi|\phi\rangle = \langle\phi|\psi\rangle\langle\psi|\phi\rangle,$$

which means that for the swap test circuit, the final probability of measuring $|0\rangle$ will be

$$Pr(|0\rangle) = \frac{1}{2} + \frac{1}{2}\langle\psi|\phi\rangle^2.$$

This probability containing the scalar product of the two states is the key for the similarity measurement. Measuring probability for state $|0\rangle$ will give a value close to $1/2$ if the dot product of $|\psi\rangle$ and $|\phi\rangle$ is close to 0, which means that these two states are orthogonal and maximally different. The measurement will give a value close to 1.0 if the dot product is close to 1.0, which means the states are almost identical.

In code, this looks quite simple. In each experiment, we construct the circuit:

```
def run_experiment(a1: np.complexfloating, a2: np.complexfloating,
                   target: float) -> None:
    """Construct swap test circuit and measure."""

    # |0> --- H --- o --- H --- Measure
    #               |
    # a1  --------- x ---------
    #               |
    # a2  ----------x ---------
    psi = state.bitstring(0) * state.qubit(a1) * state.qubit(a2)
    psi = ops.Hadamard()(psi, 0)
    psi = ops.ControlledU(0, 1, ops.Swap(1, 2))(psi)
    psi = ops.Hadamard()(psi, 0)
```

We perform the usual measurement by peek-a-boo and find the probability of qubit 0 to be in state $|0\rangle$:

```
  # Measure qubit 0 once.
  p0, _ = ops.Measure(psi, 0)
```

That's all there is to it. The variable `p0` will be the probability of qubit 0 to be found in the $|0\rangle$ state. What's left to do now is to compare this probability against a target to check that the result is valid. We allow a 5% error margin (0.05):

```
  if abs(p0 - target) > 0.05:
    raise AssertionError(
        'Probability {:.2f} off more than 5 pct from target {:.2f}'
        .format(p0, target))
  print('Similarity of a1: {:.2f}, a2: {:.2f} ==>  \%: {:.2f}'
        .format(a1, a2, 100.0 * p0))
```

and run a few experiments

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

```
def main(argv):
  [...]
  print('Swap test. 0.5 means different, 1.0 means similar')
  run_experiment(1.0, 0.0, 0.5)
  run_experiment(0.0, 1.0, 0.5)
  run_experiment(1.0, 1.0, 1.0)
  run_experiment(0.0, 0.0, 1.0)
  run_experiment(0.1, 0.9, 0.65)
  [...]
```

This should produce output similar to the following:

```
Swap test to compare state. 0.5 means different, 1.0 means similar
Similarity of a1: 1.00, a2: 0.00 ==>  %: 50.00
Similarity of a1: 0.00, a2: 1.00 ==>  %: 50.00
Similarity of a1: 1.00, a2: 1.00 ==>  %: 100.00
Similarity of a1: 0.00, a2: 0.00 ==>  %: 100.00
Similarity of a1: 0.10, a2: 0.90 ==>  %: 63.71
[...]
```

3.5 Quantum Teleportation

This section describes the quantum algorithm with one of the most intriguing names of all time – quantum teleportation (Bennett et al., 1993). It is a small example from the fascinating field of quantum information, which includes encryption and error correction. This type of algorithm exploits entanglement to communicate information across spatially separate locations.

The algorithmic story begins, as always, with Alice and Bob, placeholders for the distinct systems A and B. At the beginning of the story, they are together in a lab on Earth and create an entangled pair of qubits, for example, the Bell state $|\beta_{00}\rangle$. Let us mark the first qubit as Alice's and the second one as Bob's:

$$|\beta_{00}\rangle = \frac{|0_A 0_B\rangle + |1_A 1_B\rangle}{\sqrt{2}}$$

```
def main(argv):
  [...]

  # Step 1: Alice and Bob share an entangled pair, and separate.
  psi = bell.bell_state(0, 0)
```

After creating the state, they each take one of the qubits and physically separate – Alice goes to the Moon and Bob ships off to Mars. Let's not worry about how they are getting their super-cooled quantum qubits across the solar system. Nobody said teleportation was easy.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

Sitting there on the Moon, Alice happens to be in possession of this other qubit $|x\rangle$, which is in a specific state with probability amplitudes α and β:

$$|x\rangle = \alpha|0\rangle + \beta|1\rangle.$$

Alice does not know what the values of α and β are, and measuring the qubit would destroy the superposition. But Alice wants to communicate α and β to Bob, so that when he measures, he will obtain the basis states of $|x\rangle$ with the corresponding probabilities. How can Alice "send" or "teleport" the state of $|x\rangle$ to Bob? She can do this by exploiting the entangled qubit she already has in her hands from the time before the Moon travel.

In code, let's create the qubit $|x\rangle$ with defined values for α and β so we can check later whether Alice has teleported the correct values to Bob:

```
# Step 2: Alice wants to teleport a qubit |x> to Bob,
#         which is in the state:
#         |x> = a|0> + b|1> (with a^2 + b^2 == 1)
a = 0.6
b = math.sqrt(1.0 - a * a)
x = state.qubit(a, b)
print('Quantum Teleportation')
print('Start with EPR Pair a={:.2f}, b={:.2f}'.format(a, b))
```

Here comes the key "trick": Alice combines the new qubit $|x\rangle$ with the qubit she brought with her from Earth, the one that is entangled with Bob's qubit. We don't concern ourselves with how this might be accomplished in the physical world; we just assume it is possible:

```
# Produce combined state 'alice'.
alice = x * psi
```

Abusing notation a little bit, the combined state is now $|xAB\rangle$, with A representing Alice's entangled qubit on the Moon and B being Bob's entangled qubit on Mars. She now explicitly entangles $|x\rangle$ with the usual technique of applying a Controlled-Not gate:

```
# Alice lets qubit 0 (|x>) interact with qubit 1, which is her
# part of the entangled state with Bob.
alice = ops.Cnot(0, 1)(alice)
```

Finally, she applies a Hadamard gate to $|x\rangle$. Note that the application of an entangler circuit in reverse, with a first Controlled-Not gate followed by a Hadamard gate, is also called making a *Bell measurement*.

```
# Now she applies a Hadamard to qubit 0. Bob still owns qubit 2.
alice = ops.Hadamard()(alice, idx=0)
```

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

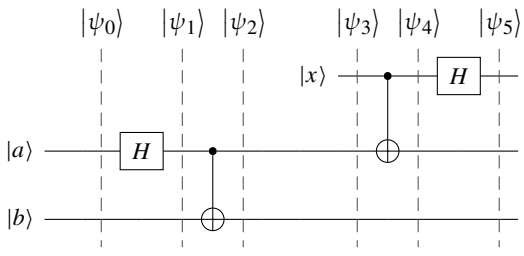


Figure 3.8 Quantum teleportation in circuit notation.

The whole procedure in circuit notation is shown in Figure 3.8. Let us analyze how the state progresses from left to right and spell out the math in great detail. Starting in the lab, before the first Hadamard gate, the state is just the tensor product of the two qubits:

$$\big|\psi_0\big\rangle = |0\rangle_A \otimes |0\rangle_B = |0_A 0_B\rangle.$$

The first Hadamard gate creates a superposition of qubit $|0\rangle_A$:

$$\big|\psi_1\big\rangle = \frac{|0\rangle_A + |1\rangle_A}{\sqrt{2}} \otimes |0\rangle_B.$$

The Controlled-Not entangles the two qubits and generates a Bell state. We discussed this mechanism in Section 2.11.2 on entanglement. Note that up to this point, Bob and Alice are at the same location: the lab on Earth.

$$\big|\psi_2\big\rangle = \frac{|0_A 0_B\rangle + |1_A 1_B\rangle}{\sqrt{2}}.$$

Alice has now traveled to the Moon, where she tensors together her new qubit $|x\rangle$ with the qubit she brought with her from Earth, resulting in state $\big|\psi_3\big\rangle$:

$$\begin{aligned}\big|\psi_3\big\rangle &= \big(\alpha|0\rangle + \beta|1\rangle\big) \otimes \frac{|0_A 0_B\rangle + |1_A 1_B\rangle}{\sqrt{2}} \\ &= \frac{\alpha|0\rangle\big(|0_A 0_B\rangle + |1_A 1_B\rangle\big) + \beta|1\rangle\big(|0_A 0_B\rangle + |1_A 1_B\rangle\big)}{\sqrt{2}}.\end{aligned}$$

Now we apply the Controlled-Not from $|x\rangle$ to Alice's qubit (this is now qubit 1). This means that the $|1\rangle$ part of the superimposed $|x\rangle$ will flip the controlled qubit. As a result, qubits $|0\rangle_A$ and $|1\rangle_A$ flip in the right-hand side of the expression:

$$\big|\psi_4\big\rangle = \frac{\alpha|0\rangle\big(|0_A 0_B\rangle + |1_A 1_B\rangle\big) + \beta|1\rangle\big(|1_A 0_B\rangle + |0_A 1_B\rangle\big)}{\sqrt{2}}.$$

Finally, we apply the Hadamard gate to $|x\rangle$, which puts $|x\rangle$'s parts $\alpha|0\rangle$ and $\beta|1\rangle$ in superposition:

$$\big|\psi_5\big\rangle = \frac{\alpha\big(|0\rangle + |1\rangle\big)\big(|0_A 0_B\rangle + |1_A 1_B\rangle\big) + \beta\big(|0\rangle - |1\rangle\big)\big(|1_A 0_B\rangle + |0_A 1_B\rangle\big)}{2}$$

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

We now multiply this out:

$$|\psi_5\rangle = \frac{1}{2}\big(\alpha(|000\rangle + |011\rangle + |100\rangle + |111\rangle) + \beta(|010\rangle + |001\rangle - |110\rangle - |101\rangle)\big)$$

We're almost there. Alice has in her possession the first two qubits. If we regroup the above expression and isolate out the first two qubits, we arrive at our target expression:

$$\begin{aligned}|\psi_5\rangle = \frac{1}{2}\big(&|00\rangle\big(\alpha|0\rangle + \beta|1\rangle\big)\\ &+ |01\rangle\big(\beta|0\rangle + \alpha|1\rangle\big)\\ &+ |10\rangle\big(\alpha|0\rangle - \beta|1\rangle\big)\\ &+ |11\rangle\big(-\beta|0\rangle + \alpha|1\rangle\big)\big).\end{aligned}$$

Remember that the first two qubits are Alice's, and the third qubit is Bob's. Alice's four basis states have probabilities determined by combinations of α and β. She can measure her first two qubits, while leaving the superposition of Bob's third qubit intact. The probability amplitudes changed, but Bob's qubit remains in superposition.

As Alice measures her two qubits, the state collapses and leaves only one probability combination for Bob's qubit. The final trick is now that Alice *tells* Bob over a classic communication channel what she has measured:

- If she measured $|00\rangle$, we know that Bob's qubit is in the state $\alpha|0\rangle + \beta|1\rangle$.
- If she measured $|01\rangle$, we know that Bob's qubit is in the state $\beta|0\rangle + \alpha|1\rangle$.
- If she measured $|10\rangle$, we know that Bob's qubit is in the state $\alpha|0\rangle - \beta|1\rangle$.
- If she measured $|11\rangle$, we know that Bob's qubit is in the state $-\beta|0\rangle + \alpha|1\rangle$.

Alice succeeded in teleporting the probability amplitudes of $|x\rangle$ to Bob. She still has to classically communicate her measurement results, so there is no faster than light communication. However, the spooky action at a distance "modified" Bob's entangled qubit on Mars to obtain the probability amplitudes from Alice's qubit $|x\rangle$, which she created on the Moon. This spooky action is truly spooky, and also astonishing.

The final step is, depending on Alice's classical communication, to apply gates to Bob's qubit to put it in the actual state of $\alpha|0\rangle + \beta|1\rangle$:

- If she sends 00, nothing needs to be done.
- If she sends 01, Bob needs to flip the amplitudes by applying the X-gate.
- If she sends 10, Bob flips the phase by applying the Z-gate.
- Correspondingly, for 11, Bob applies a Z-gate and a X-gate.

After this, Bob's qubit on Mars will be in the state of Alice's original qubit $|x\rangle$ on the Moon. Teleportation completed. Minds blown.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

In code (in file `src/teleportation.py` in the open source repository), we run four experiments, corresponding to the four possible measurement results:

```
# Alice measures and communicates the result |00>, |01>, ... to Bob.
alice_measures(alice, a, b, 0, 0)
alice_measures(alice, a, b, 0, 1)
alice_measures(alice, a, b, 1, 0)
alice_measures(alice, a, b, 1, 1)
```

For each experiment, we pretend that Alice measured a specific result and apply the corresponding decoder gates to Bob's qubit:

```
def alice_measures(alice: state.State,
            expect0: np.complexfloating, expect1: np.complexfloating,
            qubit0: np.complexfloating, qubit1: np.complexfloating):
  """Force measurement and get teleported qubit."""

  # Alices measure her state and gets a collapsed |qubit0 qubit1>.
  # She let's Bob know which one of the 4 combinations she obtained.
  # We force measurement here, collapsing to a state with the
  # first two qubits collapsed. Bob's qubit is still unmeasured.
  _, alice0 = ops.Measure(alice, 0, tostate=qubit0)
  _, alice1 = ops.Measure(alice0, 1, tostate=qubit1)

  # Depending on what was measured and communicated, Bob has to do
  # one of these things to his qubit2:
  if qubit0 == 0 and qubit1 == 0:
    pass
  if qubit0 == 0 and qubit1 == 1:
    alice1 = ops.PauliX()(alice1, idx=2)
  if qubit0 == 1 and qubit1 == 0:
    alice1 = ops.PauliZ()(alice1, idx=2)
  if qubit0 == 1 and qubit1 == 1:
    alice1 = ops.PauliX()(ops.PauliZ()(alice1, idx=2), idx=2)
```

Then, we measure Bob's qubit and confirm that it matches expectations:

```
  # Now Bob measures his qubit (2) (without collapse, so we can
  # 'measure' it twice. This is not necessary, but good to double check
  # the maths).
  p0, _ = ops.Measure(alice1, 2, tostate=0, collapse=False)
  p1, _ = ops.Measure(alice1, 2, tostate=1, collapse=False)

  # Alice should now have 'teleported' the qubit in state 'x'.
  # We sqrt() the probability, we want to show (original) amplitudes.
  bob_a = math.sqrt(p0.real)
  bob_b = math.sqrt(p1.real)
```

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

```
print('Teleported (|{:d}{:d}>)   a={:.2f}, b={:.2f}'.format(
    qubit0, qubit1, bob_a, bob_b))

if (not math.isclose(expect0, bob_a, abs_tol=1e-6) or
    not math.isclose(expect1, bob_b, abs_tol=1e-6)):
  raise AssertionError('Invalid result.')
```

This should result in the following output:

```
Quantum Teleportation
Start with EPR Pair a=0.60, b=0.80
Teleported (|00>)   a=0.60, b=0.80
Teleported (|01>)   a=0.60, b=0.80
Teleported (|10>)   a=0.60, b=0.80
Teleported (|11>)   a=0.60, b=0.80
```

The core idea of exploiting entanglement is found in other algorithms of this type. An interesting example is *superdense coding*, which we discuss next. *Entanglement swapping* would be another representative of this class of algorithms (Berry and Sanders, 2002), but we won't discuss it further here. A sample implementation can be found in file `src/entanglement_swap.py` in the open source repository.

3.6 Superdense Coding

Superdense coding, another algorithm with a very cool name, takes the core idea from quantum teleportation and turns it on its head. Alice and Bob again share an entangled pair of qubits. Alice takes hers to the Moon, while Bob takes his to Mars. Sitting on the Moon, Alice wants to communicate two classical bits to Bob. Superdense coding encodes two classical bits and sends them to Bob by physically transporting just a single qubit. Two qubits are still needed in total, but the communication is done with just a single qubit.

There exists no other classical compression scheme that would allow the compression of two classical bits into one. Of course, here we are dealing with qubits, which have two degrees of freedom (two angles define the position on a Bloch sphere). The challenge is how to exploit this fact in order to compress information. To understand how this works, we again begin with an entangled pair of qubits (the corresponding code is in file `src/superdense.py`):

```
# Step 1: Alice and Bob share an entangled pair, and separate.
psi = bell.bell_state(0, 0)
```

Alice manipulates her qubit 0 on the Moon according to the rules of how to encode two classical bits into a single qubit, as shown below. In a twist of events, she will classically ship her qubit to Bob's Mars station. There, Bob will disentangle and

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

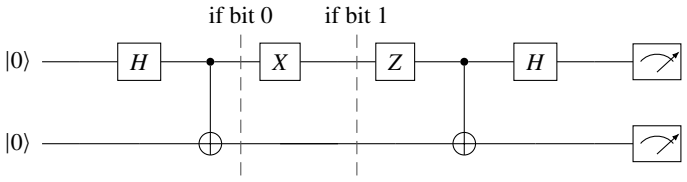


Figure 3.9 Superdense coding in circuit notation.

measure both qubits. Based on the results of the measurement, he can derive Alice's original two classical bits. Alice sent just one qubit to allow Bob to restore two classical bits.

To start the process, Alice manipulates her qubit, which is qubit 0, in the following way.

- If classical bit 0 is set, she applies the X-gate.
- If classical bit 1 is set, she applies the Z-gate.
- Of course, if both bit 0 and bit 1 are set, both the X-gate and Z-gate are being applied.

The whole procedure in circuit notation is shown in Figure 3.9. In code, the two classical bits encode four possible cases (00, 01, 10, and 11). We iterate over these four combinations to drive our experiments:

```
  # Alice manipulates her qubit and sends her 1 qubit back to Bob,
  # who measures. In the Hadamard basis he would get b00, b01, etc.
  # but we're measuring in the computational basis by reverse
  # applying Hadamard and Cnot.

  for bit0 in range(2):
    for bit1 in range(2):
      psi_alice = alice_manipulates(psi, bit0, bit1)
      bob_measures(psi_alice, bit0, bit1)
```

Here is the routine where Alice manipulates the qubits.

```
def alice_manipulates(psi: state.State,
                      bit0: int, bit1: int) -> state.State:
  """Alice encodes 2 classical bits in her 1 qubit."""

  # Note: This logic applies the Z-gate and X-gate to qubit 0.
  ret = ops.Identity(2)(psi)
  if bit0:
    ret = ops.PauliX()(ret)
  if bit1:
    ret = ops.PauliZ()(ret)
  return ret
```

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

Let us understand how the math works. The entangled pair is in the Bell state $|\beta_{00}\rangle$ initially:

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

Now let us apply the X-gate to qubit 0:

$$(X \otimes I)\,|\beta_{00}\rangle = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = |\beta_{01}\rangle.$$

Or, in short:

$$(X \otimes I)\,|\beta_{00}\rangle = \frac{|10\rangle + |01\rangle}{\sqrt{2}} = |\beta_{01}\rangle.$$

Applying the X-gate changes the state and turns it into a different Bell state – it flips the second subscript of the Bell state. This corresponds to setting the classical bit 0 to 1 in Alice's encoding.

Applying the Z-gate changes the state and flips the first subscript of the Bell state, which corresponds to setting classical bit 1 to 1 in Alice's encoding:

$$(Z \otimes I)\,|\beta_{00}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} = |\beta_{10}\rangle.$$

Applying both the X-gate and the Z-gate will change the state to $|\beta_{11}\rangle$, indicating that both classical bits 0 and 1 were set to 1. Analogously, as we saw earlier with Equation (3.2), we *could* explicitly apply iY to $|\beta_{00}\rangle$ to yield $|\beta_{11}\rangle$. Of course, this step is not needed here because the prior applications of the X-gate and Z-gate already compound to this effect.

$$(iY \otimes I)\,|\beta_{00}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}} = |\beta_{11}\rangle.$$

Bob measures in the Hadamard basis, which is another way of saying that before measurement, he converts the state to the computational basis by applying the entangler's Hadamard and Controlled-Not gates in reverse order.

Going through the entangler circuit in reverse uncomputes the entanglement and changes the state to one of the defined basis states $|00\rangle, |01\rangle, |10\rangle$, or $|11\rangle$, depending on the value of the original classical bits. The probability for each possible case is 100%. We will find the classic bit values as they were set by Alice.

```
def bob_measures(psi: state.State, expect0: int, expect1: int) -> None:
  """Bob measures both bits (in computational basis)."""

  # Change Hadamard basis back to computational basis.
  psi = ops.Cnot(0, 1)(psi)
  psi = ops.Hadamard()(psi)
```

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

```
p0, _ = ops.Measure(psi, 0, tostate=expect1)
p1, _ = ops.Measure(psi, 1, tostate=expect0)

if (not math.isclose(p0, 1.0, abs_tol=1e-6) or
    not math.isclose(p1, 1.0, abs_tol=1e-6)):
  raise AssertionError(f'Invalid Result p0 {p0} p1 {p1}')

print(f'Expected/matched: {expect0}{expect1}.')
```

This confirms results with 100% probability for the $|0\rangle$ and $|1\rangle$ states, depending on how the qubit was manipulated by Alice. Here is the expected output:

```
Expected/matched: 00
Expected/matched: 01
Expected/matched: 10
Expected/matched: 11
```

3.7 Bernstein–Vazirani Algorithm

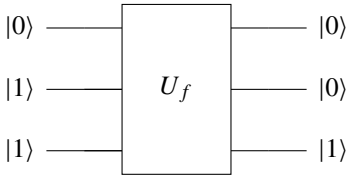
The Bernstein–Vazirani algorithm is an example for which the equivalent classical algorithm is of complexity $O(n)$, while the quantum algorithm only requires a single invocation (please note the subtle difference).

Assume we have an input state of n qubits. Also, assume there is another secret string s of the same length consisting of 0s and 1s with the property that the scalar product of the input and output bits equals 1. In other words, if the inputs are b_i and the secret string has bits s_i, then this scalar product should hold:

$$b_0 s_0 + b_1 s_1 + \cdots + b_{n-1} s_{n-1} = 1. \tag{3.4}$$

The goal is to find the secret string s. On a classical computer, we would have to try n times. Each experiment would have an input string of all 0s, except for a single 1. Each iteration where Equation (3.4) holds identifies a 1-bit in s at position t for trial $t \in [0, n-1]$.

In the quantum formulation, we construct a circuit. After the circuit has been applied, the outputs will be in states $|0\rangle$ and $|1\rangle$, corresponding to the secret string's bits, which we encode into a big unitary operator U_f. In the example below, the secret string is `001`.



https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

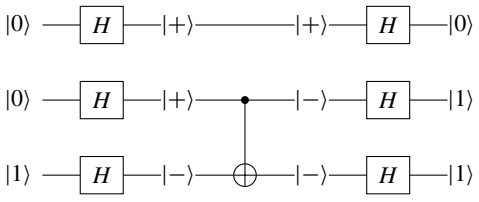


Figure 3.10 Applying Controlled-Not gates to $|+\rangle$ and $|-\rangle$.

To see how this works, we need to understand the mechanics of basis changes. Remember how the $|0\rangle$ and $|1\rangle$ states are put in superposition with Hadamard gates:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} = |+\rangle, \tag{3.5}$$

$$H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle. \tag{3.6}$$

As a first step, we create an input state of length n, initialized with all $|0\rangle$, with an additional ancilla qubit in state $|1\rangle$. The main trick of this circuit and algorithm is the following.

If we apply a Controlled-Not from a controlling qubit in the $|+\rangle$ state to a qubit in the $|-\rangle$ state, this has the effect of flipping the controlling qubit into the $|-\rangle$ state! This is the crucial trick because now applying another Hadamard gate will rotate the bases from $|+\rangle$ back to $|0\rangle$ and from $|-\rangle$ to $|1\rangle$. In other words, the qubits on which we applied the Controlled-Not will have the resulting state $|1\rangle$.

We can visualize this effect with the circuit in Figure 3.10, where we symbolically inline the states. Let us write this in code (in file `src/bernstein.py` in the open source repository). First, we create the secret string:

```
def make_c(nbits: int) -> Tuple[bool]:
  """Make a random constant c from {0,1}, the c we try to find."""

  constant_c = [0] * nbits
  for idx in range(nbits-1):
    constant_c[idx] = int(np.random.random() < 0.5)
  return tuple(constant_c)
```

Next, we construct the circuit, which is also called an *oracle*. The construction is simple – we apply a Controlled-Not for each of the qubits corresponding to 1s in the secret string. For example, for the secret string `1010`, we would construct the circuit in Figure 3.11.

We construct the corresponding circuit as one big unitary operator matrix U. This limits the number of qubits we can use but is still sufficient for exploring the algorithm.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

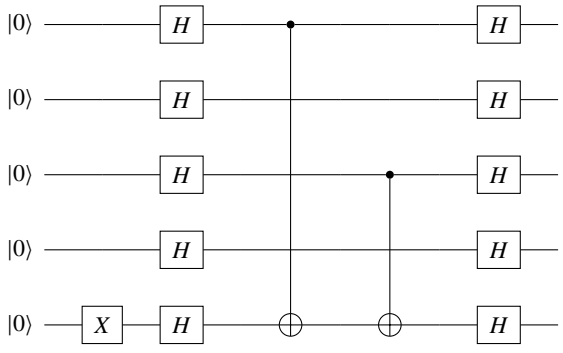


Figure 3.11 The quantum circuit for the Bernstein–Vazirani algorithm with secret string `1010`.

```
def make_u(nbits: int, constant_c: Tuple[bool]) -> ops.Operator:
  """Make general Bernstein oracle."""

  op = ops.Identity(nbits)
  for idx in range(nbits-1):
    if constant_c[idx]:
      op = ops.Identity(idx) * ops.Cnot(idx, nbits-1) @ op

  if not op.is_unitary():
    raise AssertionError('Constructed non-unitary operator.')
  return op
```

For the full circuit, we perform the following steps. First we create a secret string of length `nbits-1` and construct the big unitary. Then we build a state consisting of `nbits` states initialized as $|0\rangle$, tensored with an ancilla qubit initialized as $|1\rangle$. We follow this with the big unitary, which we sandwiched between Hadamard gates. As a final step, we measure and compare the results:

```
def run_experiment(nbits: int) -> None:
  """Run full experiment for a given number of bits."""

  c = make_c(nbits-1)
  u = make_u(nbits, c)

  psi = state.zeros(nbits-1) * state.ones(1)
  psi = ops.Hadamard(nbits)(psi)
  psi = u(psi)
  psi = ops.Hadamard(nbits)(psi)
  check_result(nbits, c, psi)
```

To check the results, we measure the probability for all possible states and ensure the state with nonzero probability ($p > 0.1$) matches the secret string. There should

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

only be one matching state of n qubits. In the code below, we iterate over all possible results and only print the results with higher probability.

```
def check_result(nbits: int, nbits: int, c: Tuple[bool],
                 psi: state.State) -> None
  """Check expected vs. achieved results."""

  print(f'Expected: {c}')

  # The state with the 'flipped' bits will have probability 1.0.
  # It will be found on the very first try.
  for bits in helper.bitprod(nbits):
    if psi.prob(*bits) > 0.1:
      print('Found   : {}, with prob: {:.1f}'
      .format(bits[:-1], psi.prob(*bits)))
      if bits[:-1] != c:
        raise AssertionError('invalid result')
```

That's it! Running this program will produce something like the following output, showing the bit settings and the resulting probabilities:

```
Expected: (0, 1, 0, 1, 0, 0)
Found   : (0, 1, 0, 1, 0, 0), with prob: 1.0
```

3.8 Deutsch's Algorithm

Deutsch's algorithm is another, somewhat contrived, algorithm with no apparent practical use (Deutsch, 1985). However, it was one of the first to showcase the potential power of quantum computers, and therefore it is always one of the first algorithms to be discussed in textbooks. Never fight the trend, let's discuss it right away.

As with the previous section on Bernstein–Vazirani, Deutsch's algorithm is in the class of "oracle" algorithms, where a larger black-box unitary operator performs a critical function. As we discuss the algorithm, it is not clear *how* the oracle is being implemented, but its function can be described, which is sufficient to show the advantages of quantum computing. The impression you get is that there is some "trick" to construct the oracle, a magical quantum way of doing this, which allows the unitary operator to answer specific algorithmic questions. This can be confusing for beginners. It is important to understand that to construct the oracle we need to visit all possible input states and explicitly construct the actual oracle to give the right answers. The oracle can be a circuit or a big permutation matrix.

The oracle is thus not a magical oracle, perhaps more *oracle adjacent*. What really makes the oracle oracle-ish is that we can feed it states in superposition. This leads to quantum parallelism, where all answers are computed in parallel. The ingenuity of a

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

particular algorithm is then to extract some meaningful information from the resulting states, which may be entangled with junk qubits.

There are a handful of oracle algorithms to be found in the literature. We will visit 2.5 of them. First, we will discuss the fundamental Deutsch algorithm, and then, later in this chapter, its extension to more than two input qubits. That's two algorithms, so we add another 1/2 algorithm by showing how to formulate the previously discussed Bernstein–Vazirani algorithm in oracle form as well, using the oracle constructor we develop in this section.

3.8.1 Problem: Distinguish Two Types of Functions

Assume we have a function f that maps a single 0 or 1 to a single 0 or 1:

$$f : \{0, 1\} \rightarrow \{0, 1\}$$

There are four possible cases for this function, which we call *constant* or *balanced*:

$$\begin{aligned} f(0) = 0, \quad f(1) = 0 \quad &\Rightarrow \text{constant}, \\ f(0) = 0, \quad f(1) = 1 \quad &\Rightarrow \text{balanced}, \\ f(0) = 1, \quad f(1) = 0 \quad &\Rightarrow \text{balanced}, \\ f(0) = 1, \quad f(1) = 1 \quad &\Rightarrow \text{constant}. \end{aligned}$$

Deutsch's algorithm answers the following question: *Given one of these four functions f, which type of function is it: balanced or constant?*

To answer this question with a classical computer, you have to evaluate the function for all possible inputs. In the quantum model, we assume we have an oracle that, given two input qubits $|x\rangle$ and $|y\rangle$, changes the state to Equation 3.7. Note that XOR, denoted by $\oplus$, is equivalent to addition modulo 2, hence the plus sign in the circle:

$$|x, y\rangle \rightarrow |x, y \oplus f(x)\rangle \tag{3.7}$$

The input $|x\rangle$ remains unmodified, $|y\rangle$ is being XOR'ed with $f(|x\rangle)$. This is a formulation that we will see in other oracle algorithms as well – there is always an ancilla $|y\rangle$, and the result of the evaluated function is XOR'ed with that ancilla. Remember that quantum operators must be reversible; this is one of the ways to achieve this.

Assuming we have oracle U_f representing and applying the unknown function $f(x)$, then the Deutsch algorithm can be drawn as the circuit shown in Figure 3.12.

As discussed, it is a convention to start every circuit with all qubits in state $|0\rangle$. The algorithm requires the ancilla qubit to be in state $|1\rangle$, which can be achieved easily by adding an X-gate to the lower qubit. Let us look at the detailed math again. Initially, after the X-gate on qubit 1, the state is:

$$|\psi_0\rangle = |01\rangle.$$

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

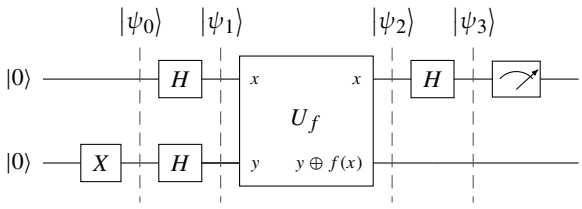


Figure 3.12 The circuit representation of Deutsch's algorithm.

After the first Hadamard gates, the state is in superposition:

$$|\psi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Applying U_f to this state's second qubit (note the operator $\oplus$ is an XOR in the right side of the equation):

$$|\psi_2\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}}.$$

If $f(x) = 0$, then $|\psi_2\rangle = |\psi_1\rangle$:

$$\begin{aligned} |\psi_2\rangle &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0 \oplus 0\rangle - |1 \oplus 0\rangle}{\sqrt{2}} \\ &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \end{aligned}$$

But if $f(x) = 1$, then

$$\begin{aligned} |\psi_2\rangle &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0 \oplus 1\rangle - |1 \oplus 1\rangle}{\sqrt{2}} \\ &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|1\rangle - |0\rangle}{\sqrt{2}}. \end{aligned}$$

We can combine the two results into a single expression:

$$|\psi_2\rangle = (-1)^{f(x)} \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

The first qubit x is in a superposition, so we multiply in the constant factor:

$$|\psi_2\rangle = \frac{(-1)^{f(x)}|0\rangle + (-1)^{f(x)}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Or, with substituting in the corresponding value for x:

$$|\psi_2\rangle = \frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \tag{3.8}$$

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

Finally, applying the final Hadamard to the top qubit takes the state from the Hadamard basis back to the computational basis. To see how this works, let us remind ourselves that the Hadamard operator is its own inverse:

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

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

If we look at Equation (3.8) and take $f(0) = f(1) = 0$, we get:

$$\begin{aligned}\big|\psi_2\big\rangle &= \frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} \\ &= \frac{(-1)^{0}|0\rangle + (-1)^{0}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} \\ &= \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.\end{aligned}$$

And then, applying the final Hadamard gate to reach state $\big|\psi_3\big\rangle$:

$$\big|\psi_3\big\rangle = H\frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |0\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

For $f(0) = f(1) = 1$ we get the same expression, but with a minus sign in front of the first qubit.

$$\big|\psi_3\big\rangle = -|0\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

For a balanced function, $f(0) = 0$ and $f(1) = 1$, and we get:

$$\begin{aligned}\big|\psi_2\big\rangle &= \frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} \\ &= \frac{(-1)^{0}|0\rangle + (-1)^{1}|1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} \\ &= \frac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.\end{aligned}$$

And then, applying the final Hadamard gate to reach state $\big|\psi_3\big\rangle$:

$$\big|\psi_3\big\rangle = H\frac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |1\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

And similarly, for $f(0) = 1$ and $f(1) = 0$, we get a similar expression, just with a minus sign in front:

$$\big|\psi_3\big\rangle = -|1\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

Table 3.2 Truth table for $f(0) = f(1) = 0$.

x	y	$f(x)$	$y \oplus f(x)$	new state
0	0	0	0	0,0
0	1	0	1	0,1
1	0	0	0	1,0
1	1	0	1	1,1

For a constant function f, we always have a $|0\rangle$ in the front, and in the balanced case the first qubit will always be in state $|1\rangle$. This means that after a single run of the circuit, we can determine the type of f simply by measuring the first qubit.

The superposition allows the computation of the results for both basis states $|0\rangle$ and $|1\rangle$ simultaneously. This is also called *quantum parallelism*. The XOR'ing to the ancilla qubit allows the math to add up in a smart way such that a result can be obtained with high probability. The result does not tell us which specific function it is out of the four possible cases, but it does tell us which of the two classes it belongs to. Because the algorithm is able to exploit superposition to compute the result in parallel, it has a true advantage over the classical equivalent.

3.8.2 Construct U_f

The math in Subsection 3.8.1 is a bit abstract, but things may become clearer when considering how to construct U_f. To reiterate, for a combined state of two qubits, the basis states are:

$$
\begin{aligned}
|00\rangle &= [1,0,0,0]^T, \\
|01\rangle &= [0,1,0,0]^T, \\
|10\rangle &= [0,0,1,0]^T, \\
|11\rangle &= [0,0,0,1]^T.
\end{aligned}
$$

We want to construct an operator that takes any linear combination of these input states to one with the condition that the second qubit flips to:

$$|x, y\rangle \rightarrow |x, y \oplus f(x)\rangle.$$

We show this by example, followed by the code to compute the oracle operator.

$f(0) = f(1) = 0$

The function f only modifies the second qubit as a function of the first qubit. For the case where $f(0) = f(1) = 0$ the truth table is shown in Table 3.2. The columns x and y represent the input qubits $f(x)$ is constant 0 in this case. The next column shows the result of XOR'ing the function's return value with y, which is $y \oplus f(x)$. The last column finally shows the resulting new state, which leaves the first qubit unmodified and changes the second qubit to the result of the previous XOR.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

Table 3.3 Truth table for $f(0) = 0$, $f(1) = 1$.

x	y	$f(x)$	$y \oplus f(x)$	new state
0	0	0	0	0,0
0	1	0	1	0,1
1	0	1	1	1,1
1	1	1	0	1,0

We can express this with a 4×4 permutation matrix, where the rows and columns are marked with the four basis states. We use the combination of x and y as row index, and the new state as column index. In this example, old state and new state are identical, and the resulting U_f matrix is the identity matrix I:

$$\begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}.$$

Note that this has to be a permutation matrix in order to make this a *reversible* operator.

$f(0) = 0, f(1) = 1$

The construction follows the same pattern as above, with a truth table as shown in Table 3.3. The table translates into this matrix:

$$\begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}.$$

$f(0) = 1, f(1) = 0$

We apply the same approach for this flavor of $f(x)$, with the truth table as shown in Table 3.4. It translates into this matrix:

$$\begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}.$$

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

Table 3.4 Truth table for $f(0) = 1$, $f(1) = 0$.

x	y	$f(x)$	$y \oplus f(x)$	new state
0	0	1	1	0,1
0	1	1	0	0,0
1	0	0	0	1,0
1	1	0	1	1,1

Table 3.5 Truth table for $f(0) = f(1) = 1$.

x	y	$f(x)$	$y \oplus f(x)$	new state
0	0	1	1	0,1
0	1	1	0	0,0
1	0	1	1	1,1
1	1	1	0	1,0

$f(0) = f(1) = 1$

For the final case, the truth table is in Table 3.5. It corresponds to this matrix:

$$\begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}.$$

3.8.3 Computing the Operator

We can compute this matrix for all cases with the help of the following code snippet (the full implementation is in `src/deutsch.py`), which mirrors how we computed the tables above. Note the `x + xor` construction for the second index below, which is a bit clumsy. We will improve upon this later.

```
def make_uf(f: Callable[[int], int]) -> ops.Operator:
  """Simple way to generate the 2-qubit, 4x4 Deutsch Oracle."""

  u = np.zeros(16).reshape(4, 4)
  for col in range(4):
    y = col & 1
    x = col & 2
    fx = f(x >> 1)
```

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

```
      xor = y ^ fx
      u[col][x + xor] = 1.0

    op = ops.Operator(u)
    if not op.is_unitary():
      raise AssertionError('Produced non-unitary operator.')
    return op
```

Now we check the four cases/matrices:

```
  for i in range(4):
    f = make_f(i)
    u = make_uf(f)
    print(f'Flavor {i:02b}: {u}')
```

which produces output like the following:

```
Flavor 00: Operator for 2-qubit state space. Tensor:
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]]
Flavor 01: Operator for 2-qubit state space. Tensor:
[[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 1.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 1.+0.j]
 [0.+0.j 0.+0.j 1.+0.j 0.+0.j]]

 [...] similar for flavors 10 and 11
```

3.8.4 Experiments

To run an experiment, we construct the circuit and measure the first qubit. If it collapses to $|0\rangle$, $f(\cdot)$ was a balanced function, according to the math above. If it collapses to $|1\rangle$, $f(\cdot)$ was a constant function.

First, we define a function `make_f` which returns a function object according to one of the four possible function flavors. We can call the returned function object as $f(0)$ or $f(1)$, it will return 0 or 1:

```
def make_f(flavor: int) -> Callable[[int], int]:
  """Return a 1-bit constant or balanced function f. 4 flavors."""

  # The 4 versions are:
  #    f(0) -> 0, f(1) -> 0  constant
  #    f(0) -> 0, f(1) -> 1  balanced
```

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

```
  #    f(0) -> 1, f(1) -> 0  balanced
  #    f(0) -> 1, f(1) -> 1  constant
  flavors = [[0, 0], [0, 1], [1, 0], [1, 1]]

  def f(bit: int) -> int:
    """Return f(bit) for one of the 4 possible function types."""
    return flavors[flavor][bit]

  return f
```

The full experiment first constructs this function object, then the oracle. Hadamard gates are applied to each qubit in an initial state $|0\rangle \otimes |1\rangle$, followed by the oracle operator and a final Hadamard gate on the top qubit:

```
def run_experiment(flavor: int) -> None:
  """Run full experiment for a given flavor of f()."""

  f = make_f(flavor)
  u = make_uf(f)
  h = ops.Hadamard()

  psi = h(state.zeros(1)) * h(state.ones(1))
  psi = u(psi)
  psi = (h * ops.Identity())(psi)
  p0, _ = ops.Measure(psi, 0, tostate=0, collapse=False)

  print('f(0) = {:.0f} f(1) = {:.0f}'
        .format(f(0), f(1)), end='')
  if math.isclose(p0, 0.0):
    print('  balanced')
    if flavor == 0 or flavor == 3:
      raise AssertionError('Invalid result, expected balanced.')
  else:
    print('  constant')
    if flavor == 1 or flavor == 2:
      raise AssertionError('Invalid result, expected constant.')
```

Finally, we check that we get the right answer for all four function flavors:

```
def main(argv):
  if len(argv) > 1:
    raise app.UsageError('Too many command-line arguments.')

  run_experiment(0)
  run_experiment(1)
  run_experiment(2)
  run_experiment(3)
```

which should result in output like this:

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

```
f(0) = 0 f(1) = 0  constant
f(0) = 0 f(1) = 1  balanced
f(0) = 1 f(1) = 0  balanced
f(0) = 1 f(1) = 1  constant
```

3.8.5 General Oracle Operator

The code above to construct the operator is *almost* general and only depends on the function f to be provided as input. A combination of basis states is being taken to another combination of basis states via a permutation matrix (only a single 1 per row and column), and the process is controlled by the function and XOR operation. In the example above, we only considered one single input qubit and one ancilla qubit (and it had that clumsy addition for the second index), but this can be easily generalized and extended to any number of input bits.

This type of oracle can be used for other algorithms. For reuse, we add this constructor function to the list of operator constructors in `lib/ops.py`.

```
def OracleUf(nbits: int, f: Callable[[List[int]], int]):
  """Make an n-qubit Oracle for function f (e.g. Deutsch, Grover)."""

  # This Oracle is constructed similar to the implementation in
  # ./deutsch.py, just with an n-bit x and a 1-bit y
  #
  dim = 2**nbits
  u = np.zeros(dim**2).reshape(dim, dim)
  for row in range(dim):
    bits = helper.val2bits(row, nbits)
    fx = f(bits[0:-1])   # f(x) without the y.
    xor = bits[-1] ^ fx

    new_bits = bits[0:-1]
    new_bits.append(xor)

    # Construct new column (int) from the new bit sequence.
    new_col = helper.bits2val(new_bits)
    u[row][new_col] = 1.0

  op = Operator(u)
  if not op.is_unitary():
    raise AssertionError('Constructed non-unitary operator.')
  return op
```

3.8.6 Bernstein–Vazirani in Oracle Form

As promised, we present the Bernstein–Vazirani algorithm in oracle form. Much of the implementation remains the same, but instead of constructing a circuit explicitly with

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

Controlled-Not gates to represent the secret number, we write an oracle function and call the `OracleUf` constructor above. This also demonstrates how a multi-qubit input can be used to build the oracle. This implementation only supports a single ancilla qubit.

First, we construct the function to compute the dot product between the state and the secret string:

```
# Alternative way to achieve the same result, using the
# Deutsch Oracle Uf.
#
def make_oracle_f(c: Tuple[bool]) -> ops.Operator:
  """Return a function computing the dot product mod 2 of bits, c."""

  const_c = c
  def f(bit_string: Tuple[int]) -> int:
    val = 0
    for idx in range(len(bit_string)):
      val += const_c[idx] * bit_string[idx]
    return val % 2
  return f
```

And then we repeat much of the original algorithm, but with an oracle:

```
def run_oracle_experiment(nbits: int) -> None:
  """Run full experiment for a given number of bits."""

  c = make_c(nbits-1)
  f = make_oracle_f(c)
  u = ops.oracleUf(nbits, f)

  psi = state.zeros(nbits-1) * state.ones(1)
  psi = ops.Hadamard(nbits)(psi)
  psi = u(psi)
  psi = ops.Hadamard(nbits)(psi)

  check_result(nbits, c, psi)
```

We run the code to convince ourselves that we implemented all this correctly:

```
Expected: (0, 1, 0, 1, 0, 0)
Found   : (0, 1, 0, 1, 0, 0), with prob: 1.0
```

3.9 Deutsch–Jozsa Algorithm

The Deutsch–Jozsa algorithm is a generalization of the Deutsch algorithm to multiple input qubits (Deutsch and Jozsa, 1992). The function to evaluate is still balanced or constant, but over an expanded domain with multiple input bits:

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

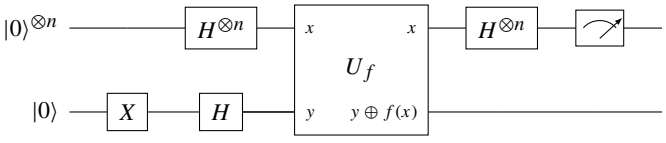


Figure 3.13 The Deutsch–Jozsa algorithm as a circuit diagram.

$$f : \{0,1\}^n \rightarrow \{0,1\}.$$

The mathematical treatment of this case parallels the two-qubit Deutsch algorithm. The key result is that we will measure a state of n qubits in the end. If we only find qubits in state $|0\rangle$ then the function is constant; if we find anything else, the function is balanced. The circuit, shown in Figure 3.13, looks similar to the two-qubit case, except that multiple qubits are being handled on both the input and output. The single ancilla qubit at the bottom will still be the key to the answer:

3.9.1 Implementation

Let us focus on the code (in file `src/deutsch_jozsa.py`), which looks quite compact with our U_f operator. First, we create the many-qubit function as either a constant function (all 0s or all 1s with equal probability) or a balanced function (the same number of 0s and 1s, randomly distributed over the length of the input bitstring). We create an array of bits and fill it with 0s and 1s accordingly. Finally, we return a function object that returns one of the values from this prepopulated array, thus representing one of the two function types:

```
def make_f(dim: int = 1,
           flavor: int = exp_constant) -> Callable[[List[int]], int]:
  """Return a constant or balanced function f over 2**dim bits."""

  power2 = 2**dim
  bits = np.zeros(power2, dtype=np.uint8)
  if flavor == exp_constant:
    bits[:] = int(np.random.random() < 0.5)
  else:
    bits[np.random.choice(power2, size=power2//2, replace=False)] = 1

  def f(bit_string:List[int]) -> int:
    """Return f(bits) for one of the 2 possible function types."""

    idx = helper.bits2val(bit_string)
    return bits[idx]

  return f
```

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

This function drives the rest of the implementation. To run an experiment, we construct the circuit shown in Figure 3.13 and measure. If the measurement finds that only state $|00\ldots0\rangle$ has a nonzero probability amplitude, then we have a constant function.

```
def run_experiment(nbits: int, flavor: int):
  """Run full experiment for a given flavor of f()."""

  f = make_f(nbits-1, flavor)
  u = ops.OracleUf(nbits, f)

  psi = (ops.Hadamard(nbits-1)(state.zeros(nbits-1)) *
         ops.Hadamard()(state.ones(1)))
  psi = u(psi)
  psi = (ops.Hadamard(nbits-1) * ops.Identity(1))(psi)

  # Measure all of |0>. If all close to 1.0, f() is constant.
  for idx in range(nbits - 1):
    p0, _ = ops.Measure(psi, idx, tostate=0, collapse=False)
    if not math.isclose(p0, 1.0, abs_tol=1e-5):
      return exp_balanced
  return exp_constant
```

Finally, we run the experiments on numbers of qubits ranging from 2 to 7 and ensure that the results match the expectations. Note that we still generate operators and oracles as full matrices, limiting the number of qubits we can handle. We will learn shortly how to expand this number by quite a bit:

```
def main(argv):
  [...]
  for qubits in range(2, 8):
    result = run_experiment(qubits, exp_constant)
    print('Found: {} ({} qubits) (expected: {})'
          .format(result, qubits, exp_constant))
    if result != exp_constant:
      raise AssertionError('Error, expected {}'.format(exp_constant))

    result = run_experiment(qubits, exp_balanced)
    print('Found: {} ({} qubits) (expected: {})'
          .format(result, qubits, exp_balanced))
    if result != exp_balanced:
      raise AssertionError('Error, expected {}'.format(exp_balanced))

if __name__ == '__main__':
  app.run(main)
```

This should result in output like the following:

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press

```
Found: constant (2 qubits) (expected: constant)
Found: balanced (2 qubits) (expected: balanced)
Found: constant (3 qubits) (expected: constant)
Found: balanced (3 qubits) (expected: balanced)
[...]
Found: constant (7 qubits) (expected: constant)
Found: balanced (7 qubits) (expected: balanced)
```

Other algorithms of this nature are *Simon's algorithm* and *Simon's generalized algorithm* (Simon, 1994). We won't discuss them here, but implementations can be found in the open-source repository in files `simon.py` and `simon_general.py`.

At this point, notice how the execution speed slows down as we increase the number of qubits in this algorithm. Once we reach 10, 11, or 12 qubits, the corresponding operator matrices become very large. Adding a few more qubits makes simulation intractable. To remedy this, we develop ways to make gate application much faster in Chapter 4, increasing our ability to simulate many more qubits.

https://doi.org/10.1017/9781009099974.005 Published online by Cambridge University Press