Contents

0.1 Additional Material for Introduction

2.3.2 Addendum: Tensoring States

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

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

But what if the states are written as an expression, such as

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

Because of the tensor product's linearity, we can actually "multiply out" the expression, just like a normal product of two terms. Correspondingly, and sometimes confusingly, the product is often written *without* the $\otimes$ operator, such as the following:

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

As we "multiply" the two bracketed terms, the scalar factors turn into simple products and the qubit states are tensored together. For scalar products the order of their operands doesn't matter (mathematically). For qubit states, on the other hand, their ordering must be maintained:

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

Writing this state as a vector results in the same state vector as above: $\begin{bmatrix} ac & ad & bc & bd \end{bmatrix}^T$. To make the required ordering clear, individual qubits sometimes get a subscript, indicating their ordering and who they belong to, such as Alice or Bob:

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

Note that the multiplication procedure can be reversed. We can *factor out* individual qubits. This should be no surprise, but it may be helpful to see this at least once:

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

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

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

2.6.3 Addendum: Pauli Representation of Operators

In the section on Pauli operators we stated that Pauli matrices form the basis for 2×2 matrices and that Hermitian density operators can be written as the following, which is also called the *Pauli representation* of an operator:

$$\rho = \frac{I + xX + yY + zZ}{2}. \tag{2.1}$$

Let's see how to derive this result. First, let's note that since we claim that the Pauli matrices form an orthonormal basis for *any* 2×2 matrix, we should be able to write any such matrix as:

$$A = cI + xX + yY + zZ. \tag{2.2}$$

By simply adding up the four matrix terms we get:

$$A = \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix}.$$

Comparing Equation 2.1 and Equation 2.2 should immediately lead to three questions:

1. Why is there no factor c in front of the I in Equation 2.1?
2. Where does the factor $1/2$ come from?
3. Given a density matrix, what are its factors x, y, and z, and maybe c?

To answer the second and third questions first. For a given state $|\psi\rangle$ and it's density matrix $\rho = |\psi\rangle\langle\psi|$, we extract the individual factors by multiplying the density matrix with the corresponding Pauli matrix and taking the trace. Let's see how this works. To extract the factor x:

$$\begin{aligned} X\,|\psi\rangle\langle\psi| &= X \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} x+iy & c-z \\ c+z & x-iy \end{bmatrix}. \end{aligned}$$

Now, taking the trace of this matrix results in:

$$\mathrm{tr}\big(X\,|\psi\rangle\langle\psi|\big) = x + iy + x - iy = 2x.$$

We are able to extract the factor x, but with a factor of 2. This is the reason

why in Equation 2.1 we compensate with a factor of 1/2. Let's derive this for the other factors, starting with Y:

$$\begin{aligned} Y\left|\psi\right\rangle\left\langle\psi\right| &= Y\begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}\begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} -i(x+iy) & -i(c-z) \\ i(c+z) & i(x-iy) \end{bmatrix} \\ &= \begin{bmatrix} -ix+y & -i(c-z) \\ i(c+z) & ix+y \end{bmatrix}. \end{aligned}$$

and

$$\operatorname{tr}\big(Y\left|\psi\right\rangle\left\langle\psi\right|\big) = -ix+y+ix+y = 2y.$$

And for Z:

$$\begin{aligned} Z\left|\psi\right\rangle\left\langle\psi\right| &= Z\begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} c+z & x-iy \\ x+iy & z-c \end{bmatrix}. \end{aligned}$$

and

$$\operatorname{tr}\big(Z\left|\psi\right\rangle\left\langle\psi\right|\big) = c+z+z-c = 2z.$$

Finally, for the identity I, the right side remains unchanged:

$$\begin{aligned} I\left|\psi\right\rangle\left\langle\psi\right| &= I\begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix} \\ &= \begin{bmatrix} c+z & x-iy \\ x+iy & c-z \end{bmatrix}. \end{aligned}$$

Taking the trace results in

$$\operatorname{tr}\big(I\left|\psi\right\rangle\left\langle\psi\right|\big) = c+z+c-z = 2c.$$

Now we make use of the fact that the trace of the density matrix of a pure state must be 1. Since we already applied a factor 1/2 in Equation 2.1, it follows that the factor c must be 1. This is why were able to omit this factor to I in Equation 2.1.

This is easy to verify in code. The full implementation is in the open-source repository in file `src/pauli_rep.py`. We construct a random qubit and extract the factors as described above:

```
qc = circuit.qc('random qubit')
qc.qubit(random.random())
qc.rx(0, math.pi * random.random())
qc.ry(0, math.pi * random.random())
qc.rz(0, math.pi * random.random())

[...]

rho = qc.psi.density()
i = np.trace(ops.Identity() @ rho)  # not strictly needed.
x = np.trace(ops.PauliX() @ rho)
y = np.trace(ops.PauliY() @ rho)
z = np.trace(ops.PauliZ() @ rho)
```

With these factors, we can verify that we indeed computed the correct results by simply plugging them into Equation 2.1:

```
new_rho = 0.5 * (i * ops.Identity() + x * ops.PauliX() +
                 y * ops.PauliY() + z * ops.PauliZ())
if not np.allclose(rho, new_rho):
  raise AssertionError('Invalid Pauli Representation')
```

Two Qubits

We computed the following for a single qubit, where σ_i are the Pauli matrices:

$$\rho = \frac{1}{2}\sum_{i=0}^{3} c_i \sigma_i.$$

This technique can be extended to two qubits by applying the same principles and multiplying the density matrix with *all* tensor products of two Pauli matrices. Similar to Equation 2.1, the density matrix can be constructed from the two-qubit bases in the following way:

$$\rho = \frac{1}{4}\sum_{i,j=0}^{3} c_{i,j} \left(\sigma_i \otimes \sigma_j\right).$$

Note that the factor is now $1/4$, or $1/2^n$ for n qubits. Also note that this form is related to the Schmidt decomposition, which we will discuss elsewhere.

We compute the factors $(c_{i,j})$ the same way as in the single-qubit case. To see how this works, let's write it in code. We first create a state and density matrix of two potentially entangled qubits:

```
qc = circuit.qc('random qubit')
qc.qubit(random.random())
qc.qubit(random.random())
```

```
# Potentially entangle them.
qc.h(0)
qc.cx(0, 1)

# Additionally rotate around randomly.
for i in range(2):
  qc.rx(i, math.pi * random.random())
  qc.ry(i, math.pi * random.random())
  qc.rz(i, math.pi * random.random())

# Compute density matrix.
rho = qc.psi.density()
```

Now we multiply with all the Pauli matrix tensor products and compute the factors from the trace. Since two qubits are involved, instead of a vector of factors (c_i) we will obtain a matrix of factors $(c_{i,j})$:

```
paulis = [ops.Identity(), ops.PauliX(), ops.PauliY(), ops.PauliZ()]
c = np.zeros((4, 4), dtype=np.complex64)
for i in range(4):
  for j in range(4):
    tprod = paulis[i] * paulis[j]
    c[i][j] = np.trace(rho @ tprod)
```

Note that in the computation of the trace above, we switched the order of `rho` and `tprod` when compared to the single-qubit case. We can do this because for two square matrices A and B, $\mathrm{tr}(AB) = \mathrm{tr}(BA)$.

Similar to above, we can now construct a new state and verify that the computed factors are correct:

```
new_rho = np.zeros((4, 4), dtype=np.complex64)
for i in range(4):
  for j in range(4):
    tprod = paulis[i] * paulis[j]
    new_rho = new_rho + c[i][j] * tprod

if not np.allclose(rho, new_rho / 4, atol=1e-5):
  raise AssertionError('Invalid Pauli Representation')
```

2.6.8 Addendum: $\pi/8$-Gate

In previous sections we discussed the Phase gate (also known as the S-gate or, in some literature, as the P-gate):

$$S = P = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/2} \end{bmatrix},$$

which represents a rotation by $\pi/2$ or 90°, given Euler's formula:

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

We can see that the S-gate is the square root of the Z-gate:

$$Z = S^2 = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/2*2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

We also learned about the T-gate, which is the root of S. It represents a rotation by half of the S-gate, which is a rotation by 45°:

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

The T-gate is sometimes called the $\pi/8$ gate, which sees counter-intuitive, given that the gate has a factor of $\pi/4$ in it! The name comes from the fact that we can make the gate more symmetric by pulling out a factor, as in:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} = e^{i\pi/8} \begin{bmatrix} e^{-i\pi/8} & 0 \\ 0 & e^{i\pi/8} \end{bmatrix}$$

2.6.9 Addendum: U-Gate

Physical quantum computers may implement other, "non-standard" types of gates. IBM machines specifically provide the general *U-gate*:

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{-i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\lambda+\phi)}\cos(\theta/2) \end{bmatrix}.$$

Note that in section 8.4.8 of the book, gate (u_3) has an error in the top right element (this has also been added to the book's errata):

$$u_3(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\lambda+\phi)}\cos(\theta/2) \end{bmatrix}‘.$$

The U-gate is quite versatile, as it can be used to construct many other standard gates [1]. For example:

$$\begin{aligned} U(\theta = \frac{\pi}{2}, \phi = 0, \lambda = \pi) &= \begin{bmatrix} \cos(\theta/2) & -e^{-i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\lambda+\phi)}\cos(\theta/2) \end{bmatrix} \\ &= \begin{bmatrix} \cos(\pi/4) & -(-1)\sin(\pi/4) \\ 1\sin(\pi/4) & -1\cos(\pi/4) \end{bmatrix}. \end{aligned}$$

With:

$$\cos\frac{\pi}{4} = \sin\frac{\pi}{4} = \frac{1}{\sqrt{2}},$$

[1] https://qiskit.org/textbook/ch-states/single-qubit-gates.html, Section 7

we are able to construct a Hadamard gate:

$$\Rightarrow U(\frac{\pi}{2}, 0, \pi) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = H$$

Another example is how to construct the flexible phase gate U_1 (with which we can generate the P-gate, S-gate, and T-gate):

$$\begin{aligned} \Rightarrow U(0, 0, \lambda) &= \begin{bmatrix} \cos(\theta/2) & -e^{-i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\lambda+\phi)}\cos(\theta/2) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix} \\ &= U_1. \end{aligned}$$

Specifically, to generate a Z-gate:

$$U(0, 0, \pi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = Z$$

Can we make an X-gate? To achieve this, we need $\cos(\theta/2) = 0$, which we can achieve by setting $\theta = \pi$. With this, the $\sin(\cdot)$ terms become 1:

$$U(\pi, \phi, \lambda) = \begin{bmatrix} 0 & -e^{-i\lambda} \\ e^{i\phi} & 0 \end{bmatrix} \tag{2.3}$$

The lower left term must be 1, hence $\phi = 0$. The upper right term must be 1 as well, hence $\lambda = \pi$. We arrive at:

$$U(\pi, 0, \pi) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = X$$

From Equation 2.3 we can also derive the Pauli Y-gate as:

$$U(\pi, \frac{\pi}{2}, \frac{\pi}{2}) = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = Y$$

The identity gate is easy to construct as

$$U(0, 0, 0) = I.$$

We can even construct rotation gates in the following way:

$$\begin{aligned} U(\theta, -\frac{\pi}{2}, \frac{\pi}{2}) &= R_x(\theta), \\ U(\theta, 0, 0) &= R_y(\theta). \end{aligned}$$

Let's verify this in code. Similar to all the other standard gates, we add this constructor to `lib/ops.py`:

```
# IBM's general U-gate.
def U(theta: float, phi: float, lam: float, d: int = 1) -> Operator:
  return Operator(np.array(
```

```
    [(np.cos(theta/2),
     -cmath.exp(1j*lam)*np.sin(theta/2)),
     (cmath.exp(1j*phi)*np.sin(theta/2),
     cmath.exp(1j*(phi+lam))*np.cos(theta/2))])).kpow(d)
```

We add a few tests to verify that our constructions were correct:

```
def test_u(self):
  val = random.random()
  self.assertTrue(np.allclose(ops.U(0, 0, val),
                              ops.U1(val)))
  self.assertTrue(np.allclose(ops.U(np.pi/2, 0, np.pi),
                              ops.Hadamard()))
  self.assertTrue(np.allclose(ops.U(0, 0, 0),
                              ops.Identity()))
  self.assertTrue(np.allclose(ops.U(np.pi, 0, np.pi),
                              ops.PauliX()))
  self.assertTrue(np.allclose(ops.U(np.pi, np.pi/2, np.pi/2),
                              ops.PauliY()))
  self.assertTrue(np.allclose(ops.U(0, 0, np.pi),
                              ops.PauliZ()))
  self.assertTrue(np.allclose(ops.U(val, -np.pi/2, np.pi/2),
                              ops.RotationX(val)))
  self.assertTrue(np.allclose(ops.U(val, 0, 0),
                              ops.RotationY(val)))
```

2.12.1 Addendum: No-Deleting Theorem

Corresponding to the just described No-Cloning Theorem, which states that in general an unknown quantum state cannot be cloned, the *No-Deleting Theorem* (Pati and Braunstein, 2000) proves that for two qubits in a random but identical state $|\psi\rangle$, there cannot be a unitary operator to *delete* or *reset* one of the two qubits back to state $|0\rangle$.

THEOREM 2.1 *Given a general quantum state $|\psi\rangle|\psi\rangle|A\rangle$, with two qubits in the identical state $|\psi\rangle$ and an ancilla $|A\rangle$, there cannot be a unitary operator U, such that $U|\psi\rangle|\psi\rangle|A\rangle = |\psi\rangle|0\rangle|A'\rangle$, where A' is the ancilla's state after application of U.*

Proof Assume we had an operator U that is capable of performing the deletion operation:

$$U|0\rangle|0\rangle A = |0\rangle|0\rangle A'$$
$$U|1\rangle|1\rangle A = |1\rangle|0\rangle A'$$

As before we compute the application of U to state $|\psi\rangle|\psi\rangle|A\rangle$ in two different

ways. First, for an individual qubit in state $\alpha|0\rangle + \beta|1\rangle$ with $|\alpha|^2 + |\beta|^2 = 1$ and with the operator U as defined above, we get:

$$\begin{aligned} U|\psi\rangle|\psi\rangle A &= |\psi\rangle|0\rangle A' \\ &= \big(\alpha|0\rangle|0\rangle + \beta|1\rangle|0\rangle\big)A' \end{aligned} \tag{2.4}$$

Now let's compute the state as the tensor product of the qubits and apply the hypothetical operator U:

$$\begin{aligned} U|\psi\rangle|\psi\rangle|A\rangle &= U\big((\alpha|0\rangle + \beta|1\rangle)(\alpha|0\rangle + \beta|1\rangle)A\big) \\ &= U\big(\alpha^2|0\rangle|0\rangle + \alpha\beta|0\rangle|1\rangle + \beta\alpha|1\rangle|0\rangle + \beta^2|1\rangle|1\rangle\big)A \\ &= \alpha^2 U|0\rangle|0\rangle A + \beta^2 U|1\rangle|1\rangle A + \alpha\beta U|0\rangle|1\rangle A + \beta\alpha U|1\rangle|0\rangle A \\ &= \alpha^2|0\rangle|0\rangle A' + \beta^2|1\rangle|0\rangle A' + \alpha\beta U\big(|0\rangle|1\rangle + |1\rangle|0\rangle\big)A \end{aligned}$$

This form is different from Equation 2.4 above. It has an additional (entangled) component $(|0\rangle|1\rangle + |1\rangle|0\rangle)$, which is typically defined as Φ. With this, the final form becomes:

$$= \big(\alpha^2|0\rangle|0\rangle + \beta^2|1\rangle|0\rangle\big)A' + \alpha\beta U\Phi A \tag{2.5}$$

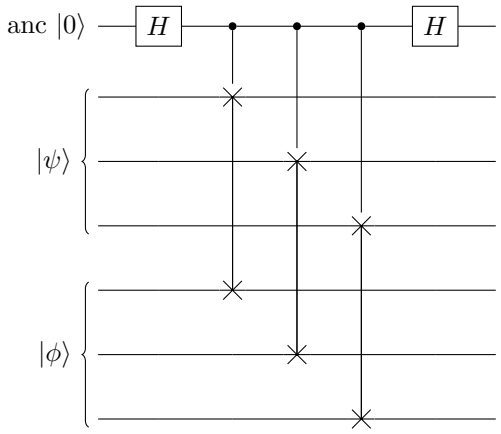
In general, Equation 2.4 is different from Equation 2.5, a contradiction which proves that no such operator U can exist. □

Note that if either $\alpha = 0$ or $\beta = 0$, we again deal with the equivalent of classical bits. The final term in Equation 2.5 would disappear and thus an operator U for those special states would indeed be feasible.

3.5 Additional Material for State Similarity

3.4.1 Swap Test for Multi-Qubit States

Now that we've learned how to use the swap test to compute the proximity of two single-qubit states, how would this work for states that are composed of multiple qubits? The answer is surprisingly simple, we just have to run multiple swap tests, for pairs of qubits, all connected to the same ancillary.



In code this looks straightforward, as shown in this snippets for two 2-qubit states:

```
def run_experiment_double(a0: np.complexfloating,
                          a1: np.complexfloating,
                          b0: np.complexfloating,
                          b1: np.complexfloating,
                          target: float) -> None:
  """Construct multi-qubit swap test circuit and measure."""

  psi_a = state.qubit(a0) * state.qubit(a1)
  psi_a = ops.Cnot(0, 1)(psi_a)
  psi_b = state.qubit(b0) * state.qubit(b1)
  psi_b = ops.Cnot(0, 1)(psi_b)

  psi = state.bitstring(0) * psi_a * psi_b

  psi = ops.Hadamard()(psi, 0)
  psi = ops.ControlledU(0, 1, ops.Swap(1, 3))(psi)
  psi = ops.ControlledU(0, 2, ops.Swap(2, 4))(psi)
  psi = ops.Hadamard()(psi, 0)

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

```
if abs(p0 - target) > 0.05:
  raise AssertionError(
      'Probability {:.2f} off more than 5% from target {:.2f}'
      .format(p0, target))
```

We can drive this implementation with a snippet like the following:

```
probs = [0.5, 0.5, 0.5, 0.52, 0.55, 0.59, 0.65, 0.72, 0.80, 0.90]
for i in range(10):
  run_experiment_double(1.0, 0.0, 0.0 + i * 0.1, 1.0 - i * 0.1,
                        probs[i])
```

3.4.2 Hadamard Test

In a previous section we discussed the *Swap Test* to measure the similarity between two unknown states $|\psi\rangle$ and $|\phi\rangle$ without having to measure these state directly. Note that we use the words *similarity* and *overlap* interchangeably. We derived that for the swap test circuit, the probability $Pr(|0\rangle)$ of measuring state $|0\rangle$ was:

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

We can invert this and express the *overlap* as:

$$\langle\psi|\phi\rangle^2 = 1 - Pr(|0\rangle).$$

In this section we present another test of this nature, the *Hadamard Test*.

Both the Swap test and the Hadamard test can be visualized with an analogy using real-valued vectors. The numbers come out differently, but the principle is the same. Think about how we compute the inner product of the sum $(\vec{a} + \vec{b})$ of two *normalized*, real-valued vectors $\vec{a}$ and $\vec{b}$ (they have to be normalized, else the math doesn't work out):

$$\begin{aligned}(\vec{a} + \vec{b})^T(\vec{a} + \vec{b}) &= \sum_i (a_i + b_i)^2 \\ &= \underbrace{\sum_i a_i^2}_{=1} + \underbrace{\sum_i b_i^2}_{=1} + 2\sum_i a_i b_i \\ &= 2 + 2\vec{a}^T\vec{b}. \end{aligned} \tag{3.6}$$

Note the three extreme cases where $\vec{a}$ and $\vec{b}$ point in the same direction, are orthogonal, or are anti-parallel. In these three cases, Equation 3.6 yields:

$$\text{parallel:} \quad \vec{a} = \begin{bmatrix}1\\0\end{bmatrix} \quad \text{and} \quad \vec{b} = \begin{bmatrix}1\\0\end{bmatrix}, \quad \text{then} \quad \vec{a}^T\vec{b} = 1.$$

$$\text{orthogonal:} \quad \vec{a} = \begin{bmatrix}0\\1\end{bmatrix} \quad \text{and} \quad \vec{b} = \begin{bmatrix}1\\0\end{bmatrix}, \quad \text{then} \quad \vec{a}^T\vec{b} = 0.$$

$$\text{anit-parallel:} \quad \vec{a} = \begin{bmatrix}-1\\0\end{bmatrix} \quad \text{and} \quad \vec{b} = \begin{bmatrix}1\\0\end{bmatrix}, \quad \text{then} \quad \vec{a}^T\vec{b} = -1.$$

Now let's apply this principle for the Hadamard test. Remember how the Swap test used two quantum registers to hold the states ψ and ϕ? The Hadamard test is different, it uses only one quantum register which will hold the superposition of the two states $|a\rangle$ and $|b\rangle$ for which we want to determine overlap. Hence, as a precondition, we need to prepare this initial state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle + |1\rangle|b\rangle\big). \tag{3.7}$$

How can we generate such a state? First, let's see how the partial expressions

look as state vectors:

$$|0\rangle|a\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}a_0\\a_1\end{bmatrix} = \begin{bmatrix}a_0\\a_1\\0\\0\end{bmatrix},$$

and

$$|1\rangle|b\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}b_0\\b_1\end{bmatrix} = \begin{bmatrix}0\\0\\b_0\\b_1\end{bmatrix}.$$

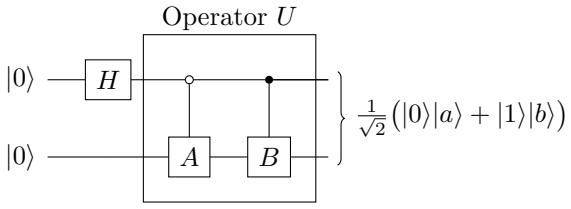
As a vector, state $|\psi\rangle$ in Equation 3.7 would be:

$$|\psi\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle + |1\rangle|b\rangle\big) = \frac{1}{\sqrt{2}}\begin{bmatrix}a_0\\a_1\\b_0\\b_1\end{bmatrix}.$$

We need operators A and B that produce the states $|a\rangle$ and $|b\rangle$ when applied to state $|0\rangle$. Note that A and B have to be a unitary and that we name the matrix element in a way to produce the desired output vectors:

$$A\,|0\rangle = A\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}a_0 & a_2\\a_1 & a_3\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}a_0\\a_1\end{bmatrix} = |a\rangle\,,$$
$$B\,|0\rangle = B\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}b_0 & b_2\\b_1 & b_3\text{‘}\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}b_0\\b_1\end{bmatrix} = |b\rangle\,.$$

To construct the circuit, we use qubit 0 as an ancilla and qubit 1 should hold the superposition of states a and b. We can use an initial Hadamard gate to produce an equal superposition of $|0\rangle$ and $|1\rangle$ with which we control the gates A and B on qubit 1, as shown in this circuit:



Note that in the literature the two controlled operators A and B are often referred to as a combined single operator U. Let's verify this in code! First we create random unitary operators A and B and apply them to state $|0\rangle$ to extract the relevant state components a_0, a_1, b_0, and b_1:

```
def make_rand_operator():
  """Make a unitary operator U, derive u0, u1."""

  U = ops.Operator(unitary_group.rvs(2))
  if not U.is_unitary():
    raise AssertionError('Error: Generated non-unitary operator')
  psi = U(state.bitstring(0))
  u0 = psi[0]
  u1 = psi[1]
  return (U, u0, u1)

def hadamard_test():
  """Perform Hadamard Test."""

  A, a0, a1 = make_rand_operator()
  B, b0, b1 = make_rand_operator()
```

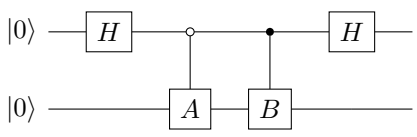
With these parameters, we can construct the state in two different ways. First we compute it explicitly, following Equation 3.7:

```
  # Construct the desired end state psi as an explicit expression.
  #    psi = 1/sqrt(2)(|0>|a> + |1>|b>)
  psi = (1 / cmath.sqrt(2) *
         (state.bitstring(0) * state.State([a0, a1]) +
          state.bitstring(1) * state.State([b0, b1])))
```

To compare, we construct the state with a circuit and confirm that the result matches the closed form above:

```
  # Let's see how to make this state with a circuit.
  qc = circuit.qc('Hadamard test - initial state construction.')
  qc.reg(2, 0)
  qc.h(0)
  qc.applyc(A, [0], 1)  # Controlled-by-0
  qc.applyc(B, 0, 1)    # Controlled-by-1

  # The two states should be identical!
  if not np.allclose(qc.psi, psi):
      raise AssertionError('Incorrect result')
```

Now let's add another Hadamard gate to the ancilla qubit 0:



This changes the state to:

$$\begin{aligned}\frac{1}{\sqrt{2}}H\big(|0\rangle|a\rangle + |1\rangle|b\rangle\big) &= \frac{1}{\sqrt{2}}\big(H|0\rangle|a\rangle + H|1\rangle|b\rangle\big)\\ &= \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle + |1\rangle|a\rangle + |0\rangle|b\rangle - |1\rangle|b\rangle\big)\\ &= \frac{1}{2}|0\rangle\big(|a\rangle + |b\rangle\big) + \frac{1}{2}|1\rangle\big(|a\rangle - |b\rangle\big).\end{aligned}$$

The probability of measuring state $|0\rangle$ in the first qubit is the norm squared of the probability amplitude:

$$\begin{aligned}\big|\frac{1}{2}\big(|a\rangle + |b\rangle\big)\big|^2 &= \frac{1}{2}\big(\langle a| + \langle b|\big)\frac{1}{2}\big(|a\rangle + |b\rangle\big)\\ &= \frac{1}{4}\big(\underbrace{\langle a|a\rangle}_{=1} + \langle a|b\rangle + \langle b|a\rangle + \underbrace{\langle b|b\rangle}_{=1}\big)\\ &= \frac{1}{4}\big(2 + \langle a|b\rangle + \langle a|b\rangle^*\big).\end{aligned} \tag{3.8}$$

The two inner products are complex numbers. For a given complex number z, adding $z + z^* = a + ib + a - ib = 2a$. Hence we can write Equation 3.8, the probability of measuring $|0\rangle$ on qubit 0, as:

$$Pr\big(|0\rangle\big) = \frac{1}{2} + \frac{1}{2}\mathrm{Re}\big(\langle a|b\rangle\big)$$

or, alternatively:

$$2Pr\big(|0\rangle\big) - 1 = \mathrm{Re}\big(\langle a|b\rangle\big)$$

We can quickly verify this in code as well:

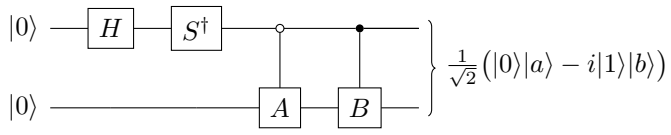
```
# Now let's apply a final Hadamard to ancilla.
qc.h(0)

# At this point, this inner product estimation should hold:
#  P(|0>) = 1/2 + 1/2 Re(<a|b>)
# Or
#  2 * P(|0>) - 1 = Re(<a|b>)
dot = np.dot(np.array([a0, a1]).conj(), np.array([b0, b1]))
p0 = qc.psi.prob(0, 0) + qc.psi.prob(0, 1)
if not np.allclose(2 * p0 - 1, dot.real, atol = 1e-6):
  raise AssertionError('Incorrect inner product estimation')
```

Can we obtain an estimate for the imaginary part of the inner product as well? Yes we can. For this, we start with a slightly modified initial state:

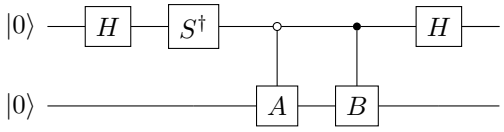
$$|\psi\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle - i|1\rangle|b\rangle\big).$$

The construction is similar to the one above, but we have to apply a factor

of $-i$ to the $|1\rangle$ part of the state by adding an $S^\dagger$ gate right after the initial Hadamard gate:



Similar to above, we add a final Hadamard gate to the ancilla:



which changes the state to:

$$\begin{aligned}\frac{1}{\sqrt{2}}H\big(|0\rangle|a\rangle - i|1\rangle|b\rangle\big) &= \frac{1}{\sqrt{2}}\big(H|0\rangle|a\rangle - Hi|1\rangle|b\rangle\big)\\ &= \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\big(|0\rangle|a\rangle + |1\rangle|a\rangle - i|0\rangle|b\rangle + i|1\rangle|b\rangle\big)\\ &= \frac{1}{2}\,|0\rangle\,\big(|a\rangle - i\,|b\rangle\big) + \frac{1}{2}\,|1\rangle\,\big(|a\rangle + i\,|b\rangle\big).\end{aligned}$$

The probability of measuring state $|0\rangle$ on the ancilla is, again, the norm squared of the probability amplitude:

$$\begin{aligned}\big|\frac{1}{2}\big(|a\rangle - i\,|b\rangle\big)\big|^2 &= \frac{1}{2}\big(\langle a| + i\,\langle b|\big)\frac{1}{2}\big(|a\rangle - i\,|b\rangle\big)\\ &= \frac{1}{4}\big(\underbrace{\langle a|a\rangle}_{=1} - i\langle a|b\rangle + i\langle b|a\rangle + \underbrace{\langle b|b\rangle}_{=1}\big)\\ &= \frac{1}{4}\big(2 - i\langle a|b\rangle + i\langle a|b\rangle^*\big).\end{aligned} \tag{3.9}$$

The inner products are complex numbers. For a complex number $z = a + ib$, $z^* = a - ib$ and these relations hold:

$$\begin{aligned}-iz &= -i(a+ib) = -ia + b,\\ iz^* &= i(a-ib) = ia + b,\\ \Rightarrow -iz + iz^* &= -ia + b + ia + b\\ &= 2b\\ &= 2\,\mathrm{Im}(z).\end{aligned}$$

Using this in Equation 3.9, we obtain the final result:

$$Pr\big(|0\rangle\big) = \frac{1}{2} + \frac{1}{2}\,\mathrm{Im}\big(\langle a|b\rangle\big).$$

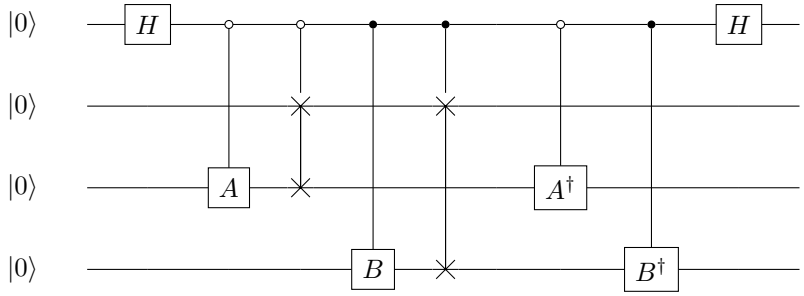
or, alternatively:

$$2Pr\big(|0\rangle\big) - 1 = \mathrm{Im}\big(\langle a|b\rangle\big).$$

We can confirm this in code as well:

```
# Now let's try the same to get to the imaginary parts.
#    psi = 1/sqrt(2)(|0>|a> - i|1>|b>)
psi = (1 / cmath.sqrt(2) *
       (state.bitstring(0) * state.State([a0, a1]) -
        1.0j * state.bitstring(1) * state.State([b0, b1])))

# Let's see how to make this state with a circuit.
qc = circuit.qc('Hadamard test - initial state construction.')
qc.reg(2, 0)
qc.h(0)
qc.sdag(0)            # <- this gate is new.
qc.applyc(A, [0], 1)  # Controlled-by-0
qc.applyc(B, 0, 1)    # Controlled-by-1

# The two states should be identical!
if not np.allclose(qc.psi, psi):
    raise AssertionError('Incorrect result')

# Now let's apply a final Hadamard to ancilla.
qc.h(0)

# At this point, this inner product estimation should hold:
#  P(|0>) = 1/2 + 1/2 Im(<a|b>)
# Or
#  2 * P(|0>) - 1 = Im(<a|b>)
dot = np.dot(np.array([a0, a1]).conj(), np.array([b0, b1]))
p0 = qc.psi.prob(0, 0) + qc.psi.prob(0, 1)
if not np.allclose(2 * p0 - 1, dot.imag, atol = 1e-6):
  raise AssertionError('Incorrect inner product estimation')
```

In cases where the construction of A and B is more complex there is the potential that the end state is entangled with the construction of A and B. In such cases, we should introduce an explicit result quantum register. After computation of A and B we superimpose the results onto this register, before uncomputing the construction of A and B, as described in the section on uncomputation. For example:


$|0\rangle$
$|0\rangle$
$|0\rangle$
$|0\rangle$
H
A
B
$A^\dagger$
$B^\dagger$
H

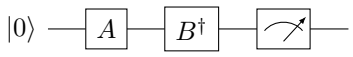
3.4.3 Inversion Test

In this chapter we will discuss the *inversion test*, a third way to approximate the similarity between states via estimating their scalar product. So far we have learned about the swap test, which utilized a register for each input $|a\rangle$ and $|b\rangle$ together with an ancilla. We also learned about the Hadamard test, which uses the ancilla but just one register, assuming there are operators A and B to construct the states $|a\rangle$ and $|b\rangle$.

The inversion test takes this one step further. It no longer needs an ancilla, just one quantum register, but it needs the ability to construct $B^\dagger$. We again assume operators A and B produce states $|a\rangle$ and $|b\rangle$, with:

$$A|0\rangle = A\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}a_0 & a_2\\ a_1 & a_3\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}a_0\\a_1\end{bmatrix} = |a\rangle\,,$$
$$B|0\rangle = B\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}b_0 & b_2\\ b_1 & b_3\text{‘}\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}b_0\\b_1\end{bmatrix} = |b\rangle\,,$$

and construct this simple circuit:



The expectation value of the projective measurement $M = |0\rangle\langle 0|$ is given by

$$\begin{aligned}\big(\langle 0|A^\dagger B|0\rangle\big)\big(\langle 0|B^\dagger A|0\rangle\big) &= \langle 0|A^\dagger B|0\rangle\langle 0|B^\dagger A|0\rangle\\ &= |\langle 0|B^\dagger A|0\rangle|^2\\ &= |\underbrace{\langle 0|B^\dagger}_{=\langle b|}\underbrace{A|0\rangle}_{=|a\rangle}|^2\\ &= |\langle b|a\rangle|^2\\ &= \langle a|b\rangle\langle b|a\rangle = \langle b|a\rangle\langle a|b\rangle\\ &= |\langle a|b\rangle|^2.\end{aligned}$$

Note that the norm of the inner product is symmetric. In code, we reuse the mechanism introduced in the section on the Hadamard test to construct random unitaries A and B with function `make_rand_operator()`. The inversion test itself is then straightforward:

```
def inversion_test():
  """Perform Inversion Test."""

  # The inversion test allows keeping the number of qubits to a minimum
  # when trying to determine the overlap between two states. However, it
  # requires a precise way to generate states a and b, as well as the
  # adjoint for one of them.
  #
```

```
# If we have operators A and B (similar to the Hadamard Test),
# to determine the overlap between a and be (<a|b>), we run:
#     B_adjoint A |0>
# and determine the probability p0 of measuring |0>. p0 is an
# a precise estimate for <a|b>.

A, a0, a1 = make_rand_operator()
B, b0, b1 = make_rand_operator()

# For the inversion test, we will need B^\dagger:
Bdag = B.adjoint()

# Compute the dot product <a|b>:
dot = np.dot(np.array([a0, a1]).conj(), np.array([b0, b1]))

# Here is the inversion test. We run B^\dagger A |0> and find
# the probability of measuring |0>:
qc = circuit.qc('Hadamard test - initial state construction.')
qc.reg(1, 0)
qc.apply1(A, 0)
qc.apply1(Bdag, 0)

# The probability amplitude of measuring |0> should be the
# same value as the dot product.
p0, _ = qc.measure_bit(0, 0)
if not np.allclose(dot.conj() * dot, p0):
  raise AssertionError('Incorrect inner product estimation')
```

3.4.4 Euclidean Distance

The core mechanism of the swap test can be used to compute the Euclidean distance between two arbitrary vectors $\vec{A}$ and $\vec{B}$. Classically, this distance is computed as the norm of a vector difference:

$$D = |\vec{A} - \vec{B}|$$

The quantum technique assumes that encoding the input vectors as states is feasible and, in many algorithm evaluations, a zero-cost process, which may be hard to achieve in practice. Here, without loss of generality, we assume that the vectors have dimensions that are a power of 2. A simple Python example would be:

```
a = [3, 5, 8, 7]
b = [1, 8, 7, 8]
```

In a first step we convert the vectors to states via amplitude encoding. We compute each vector's norm, $|\vec{A}|$ and $|\vec{B}|$, and divide each vector element by the norm. Again, we assume this is possible in practice:

$$\vec{A} \rightarrow |A\rangle = \frac{1}{|\vec{A}|} \sum_i A_i |i\rangle$$

$$\vec{B} \rightarrow |B\rangle = \frac{1}{|\vec{B}|} \sum_i B_i |i\rangle$$

Note that we can get the original vector $\vec{A}$ back by multiplying the normalized state $|A\rangle$ with its norm, as in $\vec{A} = |\vec{A}|\,|A\rangle$. In code, normalizing the vectors is quite easy with `numpy`:

```
def run_experiment(a, b):
  """Compute Euclidian Distance between vectors a and b."""

  norm_a = np.linalg.norm(a)
  norm_b = np.linalg.norm(b)
  if norm_a == 0 or norm_b == 0:
    return
  normed_a = a / norm_a
  normed_b = b / norm_b
```

Next, we cleverly construct two states $|\phi\rangle$ and $|\psi\rangle$ in the following way:

$$|\phi\rangle = \frac{1}{\sqrt{Z}}\big(|\vec{A}|\,|0\rangle - |\vec{B}|\,|1\rangle\big)$$

$$|\psi\rangle = \frac{1}{\sqrt{2}}\big(|0\rangle \otimes |A\rangle + |1\rangle \otimes |B\rangle\big)$$

with $Z = |\vec{A}|^2 + |\vec{B}|^2$. To construct the state with a circuit we can borrow the ideas from the Hadamard test. Here we construct the state directly:

```
# Create state phi:
#  |phi> = 1 / sqrt(Z) (|a| |0> - |b| |1>)
phi = state.State(1 / np.sqrt(Z) * np.array([norm_a, -norm_b]))

# Create state psi:
#  |psi> = 1 / sqrt(2) |0>|a> + |1>|b>)
psi = (state.bitstring(0) * state.State(normed_a) +
       state.bitstring(1) * state.State(normed_b)) / np.sqrt(2)
```

Finally, similar to the swap test, we create a state `combo` consisting of an ancilla initialized as $|0\rangle$ tensored with the states $|\phi\rangle$ and $|\psi\rangle$:

```
# Make a combined state with an ancilla (|0>), phi, and psi:
combo = state.bitstring(0) * phi * psi
```

We implement the swap test circuit and measure the probability of the ancilla to be $P(|0\rangle)$. Note that while state $|\psi\rangle$ has multiple qubits, we only connect the controlled swap gate to $|\phi\rangle$ and the top qubit of $|\psi\rangle$:

```
# Construct a swap test and find the measurement probability
# of the ancilla.
combo = ops.Hadamard()(combo, 0)
combo = ops.ControlledU(0, 1, ops.Swap(1, 2))(combo)
combo = ops.Hadamard()(combo, 0)

p0, _ = ops.Measure(combo, 0)
```

From the swap test we know that the probability of measuring a $|0\rangle$ for the ancilla is:

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

Let's compute the inner product $\langle\phi|\psi\rangle$ of the two states and use the fact that for a scalar/vector tensor product, it holds that $k \otimes \vec{A} = k\vec{A}$:

$$\begin{aligned}
&\langle\phi|\psi\rangle \qquad (3.10)\\
&= \big\langle \frac{1}{\sqrt{Z}}\big(|\vec{A}|\,|0\rangle - |\vec{B}|\,|1\rangle\big) \;|\; \frac{1}{\sqrt{2}}\big(|0\rangle\otimes|A\rangle + |1\rangle\otimes|B\rangle\big)\big\rangle\\
&= \frac{1}{\sqrt{Z}}\big(|\vec{A}|\,\langle 0| - |\vec{B}|\,\langle 1|\big)\frac{1}{\sqrt{2}}\big(|0\rangle\otimes|A\rangle + |1\rangle\otimes|B\rangle\big)\\
&= \frac{1}{\sqrt{2Z}}\big(|\vec{A}|\underbrace{\langle 0|\,|0\rangle}_{=1}\otimes|A\rangle + |\vec{A}|\underbrace{\langle 0|\,|1\rangle}_{=0}\otimes|B\rangle - |\vec{B}|\underbrace{\langle 1|\,|0\rangle}_{=0}\otimes|A\rangle - |\vec{B}|\underbrace{\langle 1|\,|1\rangle}_{=1}\otimes|B\rangle\big)
\end{aligned}$$

$$= \frac{1}{\sqrt{2Z}}\big(|\vec{A}| \otimes |A\rangle - |\vec{B}| \otimes |B\rangle\big)$$
$$= \frac{1}{\sqrt{2Z}}\big(|\vec{A}|\,|A\rangle - |\vec{B}|\,|B\rangle\big) \tag{3.11}$$

From the swap test we know that the probability of measuring a $|0\rangle$ for the ancilla is

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

Substituting in Equation 3.11 to compute the Euclidean distance D:

$$\begin{aligned} 2(P(|0\rangle) - \frac{1}{2}) &= |\langle\phi|\psi\rangle|^2 \\ &= \frac{1}{2Z}\big||\vec{A}|\,|A\rangle - |\vec{B}|\,|B\rangle\big|^2 \\ &= \frac{1}{2Z}|\vec{A} - \vec{B}|^2 \\ &= \frac{1}{2Z}D^2 \end{aligned}$$

The Euclidean distance D is then:

$$D = \sqrt{4Z\big(P(|0\rangle) - \frac{1}{2}\big)}$$

In code we can compute D and also compute the Euclidean distance classically to allow comparison of the results:

```
# Now compute the Euclidean norm from the probability.
eucl_dist_q = (4 * Z * (p0 - 0.5)) ** 0.5

# We can also compute the Euclidean distance classically.
eucl_dist_c = np.linalg.norm(a - b)

if not np.allclose(eucl_dist_q, eucl_dist_c):
  raise AssertionError('Incorrect computation')
```

To verify, we can drive this code in ways similar to this:

```
for iter in range(10):
  a = np.array(random.choices(range(100), k=8))
  b = np.array(random.choices(range(100), k=8))
  run_experiment(a, b)
```

6.16 Additional Algorithms

6.16.1 3-SAT

TODO

6.16.2 Graph Coloring

TODO

6.16.3 HHL Algorithm

The Harrow-Hassidim-Lloyd algorithm (HHL) solves for $\vec{x}$ in a system of linear equations:

$$A\vec{x} = \vec{b}. \tag{6.12}$$

This is also called a *linear system problem* (LSP). For example, for 3 equations with 3 variables x, y, and z, this system of equations:

$$\begin{aligned} 2x + 2y - 1z &= 3, \\ x - 3y + 4z &= 4, \\ -x + 1y - 2z &= -5. \end{aligned}$$

can be written in matrix form:

$$\begin{bmatrix} 2 & 2 & -1 \\ 1 & -3 & 4 \\ -1 & 1 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ -5 \end{bmatrix}.$$

The solution to this linear system would be $x = 1$, $y = 2$, and $z = 3$. Problems like this can of course be generalized:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_1 + \cdots + a_{1,n}x_n &= b_1, \\ a_{2,1}x_2 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2, \\ &\vdots \\ a_{n,1}x_n + a_{n,2}x_n + \cdots + a_{n,n}x_n &= b_n. \end{aligned}$$

Classically, this problem is solved using Gaussian elimination, which is of complexity $O(n^3)$. Depending on the type of matrix, however, faster techniques may apply. Let's introduce two matrix properties:

- A matrix is *s-sparse*, if it has at most s non-zero entries in each row and column. Typically, *s-sparse vectors* are defined as having only s non-zero elements, so s-sparse matrices have s-sparse vectors as rows and columns. Note that in the literature, the definition of sparsity may only pertain to either rows or columns, but here we require s-sparsity for both.

- A matrix *condition number* κ is a metric on how *invertible* a given matrix is and is usually calculated as $\kappa = ||A|| \cdot ||A^{-1}||$. It hints at the accuracy that can be obtained when solving an LSP. A large condition number points to high sensitivity - small changes in $\vec{b}$ may result in large errors in $\vec{x}$. For Hermitian matrices, κ can be defined as the ratio of the largest to smallest eigenvalue (which is undefined if the denominator is 0).

For the special case of sparse matrices and a given approximation accuracy of ϵ, solving the LSP of size N classically with the conjugate gradient method has a complexity of $O(Ns\kappa \log(1/\epsilon))$. The quantum HHL algorithm, on the other hand, is of complexity $O(\log(N)s^2\kappa^2/\epsilon)$. This presents an exponential speedup over the size N of the system, but a polynomial slowdown in s and κ. To a degree, even the exponential speedup is questionable because the vector $\vec{b}$ has to be encoded as a state, which can be costly, and reading out a full result would result in additional complexity of $O(N)$. The algorithm will only be applicable in cases where sampled results suffice. This will become more clear below.

In the quantum case, the matrix A must be a $N \times N$ Hermitian matrix, with (for simplicity) N being of the form 2^n. $\vec{x}$ and $\vec{b}$ are N-dimensional vectors. A and $\vec{b}$ are known, $\vec{x}$ is the unknown we are trying to solve for:

$$\vec{x} = A^{-1}\vec{b}. \tag{6.13}$$

Let's recall the required linear algebra. A complex square matrix A is *normal* if it commutes with its conjugate transpose:

$$A^\dagger A = AA^\dagger. \tag{6.14}$$

It can be shown that A is normal only if it is unitarily diagonalizable, which means there exists a unitary matrix U such that:

$$A = UDU^\dagger. \tag{6.15}$$

where D is a diagonal matrix. The diagonal elements of D will hold the eigenvalues of A, which must be real if A is Hermitian; the columns of U will be the orthonormal eigenvectors of A.

All unitary and Hermitian matrices are normal. This means that the *finite-dimensional spectral theorem* applies, which states that normal matrices can be written as the sum of the products of the eigenvalues λ_i with the outer product of the eigenvectors u_i. This is really just a different way to write Equation 6.15:

$$A = \sum_{i=0}^{N-1} \lambda_i |u_i\rangle\langle u_i|. \tag{6.16}$$

In this form, it is trivial to compute A's inverse as:

$$A^{-1} = \sum_{i=0}^{N-1} \lambda_i^{-1} |u_i\rangle\langle u_i|. \tag{6.17}$$

Let us explore these concepts in code (in file `src/spectral_decomp.py`).

First we generate a random unitary U, which may not be Hermitian. We make it Hermitian by computing $1/2(A+A^\dagger)$ (which may no longer be a unitary matrix):

```
u = scipy.stats.unitary_group.rvs(ndim)
umat = ops.Operator(u)
hmat = 0.5 * (umat + umat.adjoint())
```

We compute the eigenvalues and eigenvectors and check that the eigenvectors are real and orthonormal:

```
w, v = np.linalg.eig(hmat)

# Check that the eigenvalues are real.
for i in range(ndim):
  if not np.allclose(w[i].imag, 0.0):
    raise AssertionError('Found non-real eigenvalue.')

# Check that the eigenvectors are orthogonal.
for i in range(ndim):
  for j in range(i+1, ndim):
    dot = np.dot(v[:, i], v[:, j].adjoint())
    if not np.allclose(dot, 0.0, atol=1e-5):
      raise AssertionError('Invalid, non-orthogonal basis found')

# Check that eigenvectors are normal.
for i in range(ndim):
  dot = np.dot(v[:, i], v[:, i].adjoint())
  if not np.allclose(dot, 1.0, atol=1e-5):
    raise AssertionError('Found non-orthonormal basis vectors')
```

We then compute a matrix using the spectral theorem as shown in Equation 6.16 and verify that it indeed matches the original matrix before decomposition:

```
x = np.matrix(np.zeros((ndim, ndim)))
for i in range(ndim):
  x = x + w[i] * np.outer(v[:, i], v[:, i].adjoint())
if not np.allclose(hmat, x, atol=1e-5):
  raise AssertionError('Spectral decomp doesn\'t seem to work.')
```

And finally, we also check the construction of the inverse matrix by only computing the inverse eigenvalues in Equation 6.17 matches the computation of the inverse:

```
x = np.matrix(np.zeros((ndim, ndim)))
for i in range(ndim):
```

```
    x = x + 1 / w[i] * np.outer(v[:, i], v[:, i].adjoint())
  if not np.allclose(np.linalg.inv(hmat), x, atol=1e-5):
    raise AssertionError('Inverse computation doesn\'t seem to work.')
```

From above we know that complex $N \times N$ Hermitian matrices have N linearly independent, orthogonal basis vectors with real eigenvalues. This means that *any* vector in $\mathbb{C}^N$ can be constructed from such orthogonal basis. Hence we can write $\vec{b}$ as a linear combination of A's basis vectors $|u_i\rangle$, with:

$$|b\rangle = \sum_{i=0}^{N-1} b_i \, |u_i\rangle \, . \tag{6.18}$$

We combine this with Equation 6.13 and Equation 6.17 as:

$$\begin{aligned} |x\rangle &= A^{-1} \, |b\rangle \\ &= \sum_{i=0}^{N-1} \lambda_i^{-1} |u_i\rangle\langle u_i| \; b_i \, |u_i\rangle \\ &= \sum_{i=0}^{N-1} \lambda_i^{-1} b_i |u_i\rangle \underbrace{\langle u_i|u_i\rangle}_{=1} \\ \Rightarrow |x\rangle &= \sum_{i=0}^{N-1} \lambda_i^{-1} b_i |u_i\rangle. \end{aligned} \tag{6.19}$$

Equation 6.19 is the form we will use in the HHL algorithm to solve for $|x\rangle$.

6.16.4 Rotation About Y-Axis

Let's briefly talk about *rotation about the y-axis*, which is of importance in the derivation of the algorithm. Given a single-qubit state $|\psi\rangle$ and a factor $|\alpha| \leq 1.0$, we know that we can write the state in the following form because adding up the norm of the probability amplitudes will add up to 1.0:

$$|\psi\rangle = \sqrt{1 - \alpha^2} \; |0\rangle + \alpha \; |1\rangle \, . \tag{6.20}$$

In the following, we show how this can be achieved by a rotation about the y-axis by a specific angle θ.

In the HHL algorithm, the α in Equation (6.20) will be of the form $\alpha = C/\lambda$. The factor C is a normalization constant and λ is an eigenvalue that we are interested in. Note that because we compute the square root $\sqrt{1 - \alpha^2}$, α must be smaller than 1 and hence the factor C must be smaller than λ:

$$0 \leq \alpha \leq 1 \Rightarrow C \leq \lambda.$$

Let us remind ourselves of the R_y operator to perform a rotation about the

y-axis:

$$R_y(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}.$$

Applying R_y to state $|0\rangle$:

$$\begin{aligned} R_y(\theta)\,|0\rangle &= R_y(\theta) \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} \cos\frac{\theta}{2} \\ \sin\frac{\theta}{2} \end{bmatrix} \\ &= \begin{bmatrix} \sqrt{1 - \sin^2\frac{\theta}{2}} \\ \sin\frac{\theta}{2} \end{bmatrix}. \end{aligned}$$

If we now define

$$\theta = 2\,\arcsin(C/\lambda),$$

then

$$R_y(\theta)\,|0\rangle = \sqrt{1 - \frac{C^2}{\lambda^2}}\,|0\rangle + \frac{C}{\lambda}\,|1\rangle\,,$$

and with $C = 1$:

$$R_y(\theta)\,|0\rangle = \sqrt{1 - \frac{1}{\lambda^2}}\,|0\rangle + \frac{1}{\lambda}\,|1\rangle\,.$$

This is the form we saw above in Equation (6.20) with $\alpha = 1/\lambda$. If we create and measure this state multiple times, the probability of measuring $|1\rangle$ will approximate $|1/\lambda|^2$, the norm squared of the probability amplitude $1/\lambda$. This is one of the tricks in the HHL algorithm.

We can easily test this out in code. For a given value `lam`, we compute the result in two different ways and make sure the results match:

```
factor_0 = np.sqrt(1 - 1.0 / (lam * lam))
factor_1 = 1.0 / lam

# Compute a y-rotation by theta:
#
theta = 2.0 * np.arcsin(1.0 / lam)
psi = state.zeros(1)
psi = ops.RotationY(theta)(psi)

if not np.isclose(factor_0, psi[0], atol=0.001):
  raise AssertionError('Invalid computation.)')
if not np.isclose(factor_1, psi[1], atol=0.001):
  raise AssertionError('Invalid computation.)')
```

6.16.5 Vector Encoding

Linear system problems deal with vector and matrices. For the HHL algorithm, we need a method to represent those in a quantum way. For complex vectors, there are typically two different encoding schemes.

In *amplitude encoding*, vector values are simple expressed as probability amplitudes. We have to make sure that the sum of the probabilities computed from the squared norms of the amplitudes adds up to one. In our infrastructure, this can be achieved with a simple normalization step. We also want to ensure that vectors are padded to lengths that are a power of two.

A typical example given in the literature is this vector of three elements:

$$[0.1 \; -0.7 \;\; 1.0].$$

We pad it to 4 elements (the next power of two):

$$[0.1 \; -0.7 \;\; 1.0 \;\; 0.0],$$

and normalize it, which, in code, is straightforward:

```
def amplitude_encoding():
  psi = state.State([0.1, -0.7, 1.0, 0.0])
  psi.normalize()
  prob = 0.0
  for i in range(4):
    prob += psi[i] * psi[i].conj()
  if not np.allclose(prob, 1.0, atol=0.00001):
    raise AssertionError('Invalid probability amplitudes', sum)
  psi.dump()
>>
|00> (|0>):  ampl: +0.08+0.00j prob: 0.01 Phase:   0.0
|01> (|1>):  ampl: -0.57+0.00j prob: 0.33 Phase: 180.0
|10> (|2>):  ampl: +0.82+0.00j prob: 0.67 Phase:   0.0
```

Vectors can also be encoded in a scheme called *basis encoding*. In this scheme, depending on the required accuracy, floating point numbers are encoded as binary fractions, with qubits representing fractional powers of 2. We have seen how to encode binary fractions with states as:

$$|\psi\rangle = |x_0 \; x_1 \; \cdots \; x_{n-2} \; x_{n-1}\rangle.$$

The x_i should be interpreted as binary bits, with values of either 0 or 1. This looks natural in the following mathematical notation (note that the bit ordering is up to convention and can be reversed):

$$\frac{x_0}{2^1} = x_0 \frac{1}{2^1} = 0.x_0,$$
$$\frac{x_0}{2^1} + \frac{x_1}{2^2} = x_0 \frac{1}{2^1} + x_1 \frac{1}{2^2} = 0.x_0 x_1,$$

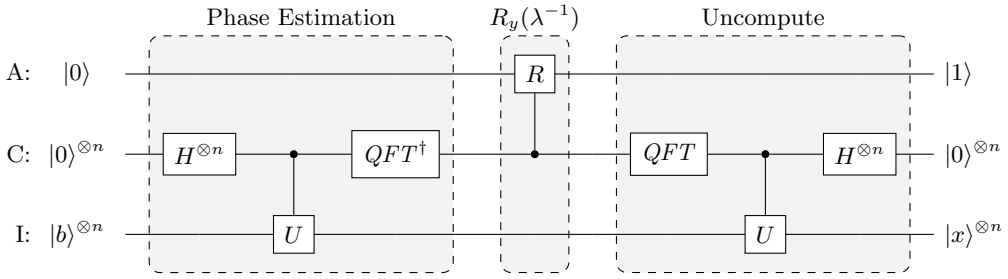


Figure 6.1 The HHL algorithm as a circuit diagram.

$$\frac{x_0}{2^1} + \frac{x_1}{2^2} + \frac{x_2}{2^3} = x_0\frac{1}{2^1} + x_1\frac{1}{2^2} + x_2\frac{1}{2^3} = 0.x_0x_1x_2,$$
$$\ldots$$

If we decide to reserve, say, 5 qubits to represent a binary fraction (1 sign bit and 4 bits to represent the value), then the vector $[0.1, -0.7, 1.0, 0.0]$ from above can be approximated as:

$$\begin{aligned} 0.082 &\simeq |00001\rangle = 0.0635, \\ -0.7 &\simeq |11011\rangle = -0.688, \\ 1.0 &\simeq |01111\rangle = 0.936. \end{aligned}$$

To represent the vector of the three approximated values from above, we simply create a state of 15 qubits, concatenating the states from the basis encoding. In the example, the encoded state would be:

$$|\psi\rangle = |000011101101111\rangle$$

With this type of encoding, the vector doesn't need to be normalized to $[0.082, -0.57, 0.82]$. However, the largest value that can be approximated is 1.0, which means the individual vector elements must be scaled appropriately.

6.16.6 Algorithm

The HHL algorithm has the following five major components, as shown in Figure 6.1:

1. State Preparation (not shown in the Figure). The vector $\vec{b}$ needs to be encoded as a quantum state. This function is critical - not all states can be encoded easy as quantum states. Complications in this step may reduce or even eliminate the algorithm's quantum advantage.
2. Quantum phase estimation.
3. Ancilla qubit rotation.
4. Uncomputation.
5. Measurement and interpretation of the results.

Let us assume vector $\vec{b}$ has N_b components (which should be a power of 2). We use amplitude encoding to encode this vector as a quantum state, which means we need n_b qubits, with $n_b = log(N_b)$. The corresponding quantum circuit consists of three register files.

A single *ancilla* qubit A. When written as a (sub-)state, we will denote this qubit with $|\,\rangle_a$.

A *clock register* C of width n. This register will contain the encoded eigenvalue coming out of the quantum phase estimation. More registers provide higher accuracy. In state notation, we refer to this register as $|\,\rangle_c$.

An *input register* I of width n_b, which will contain the results of the state preparation, a representation of the initial vector $\vec{b}$. Similar to above, we denote this register state as $|\,\rangle_b$.

State Preparation

We use amplitude encoding as outlined in Section Section 6.16.5. We assume this step is possible and efficient on a physical system (which, again, might not be the case for all possible vectors). With the notation shown above, the system after state preparation will be in the following state, with $|b\rangle_b$ indicating a register of size n_b:

$$|\psi\rangle = |0\rangle_a \, |0 \cdots 00\rangle_c \, |b\rangle_b$$

Bibliography

A. Pati and S. Braunstein. Impossibility of deleting an unknown quantum state. *Nature*, 404(6774):164–165, Mar. 2000. ISSN 0028-0836.