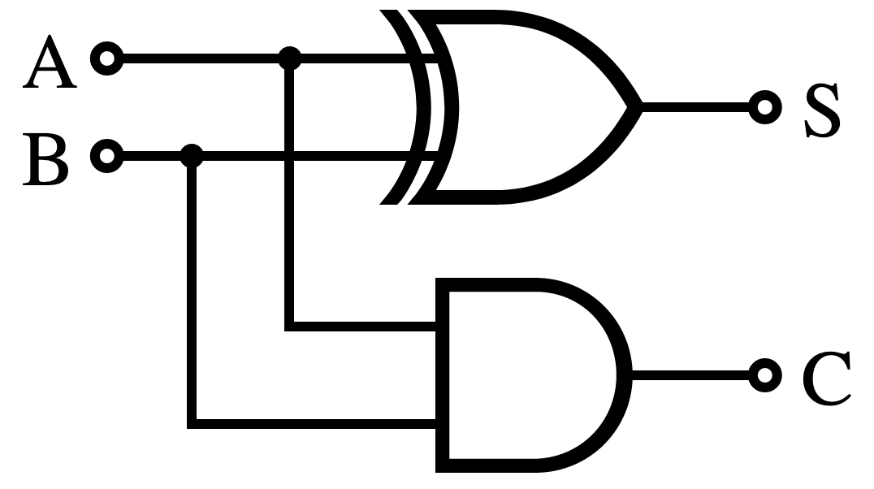
**Introduction to Qubits and Quantum States**

**The Atoms of Computation**

Bits are designed to be the world’s most basic language. They can represent any information with just 1s and 0s. For example, the number 9213 can be written in binary as 10001111111101. Strings like these are known as binary strings and can represent text as well as bits. Binary conversions for all characters can be found in [this table](https://www.ibm.com/docs/en/aix/7.3?topic=adapters-ascii-decimal-hexadecimal-octal-binary-conversion-table). This is simply a widely agreed upon standard, and individual assignments are highly arbitrary.

Quantum computers process information in qubits, instead of in bits. For this first section, qubits will be treated identically to bits, and quantum mechanical processes will be considered later on.

All types of bits must be manipulated in order to turn inputs into outputs. For the simplest programs, this can be done as a circuit diagram. These depict inputs, outputs, and gates between the two. Here’s an example for a standard computer:



For quantum circuits, the same process is followed but different conventions are used to represent inputs, outputs, and operators. Here’s the same circuit as a quantum circuit diagram:

A diagram of a diagram

Description automatically generated

In a circuit, we have to do 3 jobs: encode the input, perform computation, and extract an output. Let’s begin by focusing on the last of these jobs. The following code encodes a quantum circuit with 8 qubits and 8 outputs:

qc\_output = QuantumCircuit(8)

This circuit, which we have called qc\_output, is created by Qiskit using QuantumCircuit. The QuantumCircuit takes the number of qubits in the quantum circuit as an argument.

The extraction of outputs in a quantum circuit is done by the operation measure\_all(). Each measurement tells the qubit to output a specific bit.

qc\_output.measure\_all()

This segment of code adds a measurement to each qubit in qc\_output and writes it to a classical bit.

Now that our circuit has something in it, let’s take a look at it.

qc\_output.draw(initial\_state=True)

A graph with black squares and arrows

Description automatically generated

Qubits are always initialized to output the 0 state. Since we haven’t done anything, this is what we measure. We can see this by running our system many times:

sim = Aer.get\_backend(‘aer\_simulator’)

result = sim.run(qc\_output).result()

counts = result.get\_counts()

plot\_histogram(counts)

A blue rectangular object with numbers and lines

Description automatically generated

We must run the circuit many times to be certain, since quantum processes may have some randomness in their results. Since we’re not doing anything quantum here, we get the result 0 with certainty.

This result comes from a classical computer simulating what a quantum computer would do. Quantum simulations are only really possible for a small number of qubits, generally under 30, but they can be powerful tools for designing quantum circuits. To run on a real device, you would replace Aer.get\_backend(‘aer\_simulator’) with the backend of the object you want to run on.

Let’s look at how to encode a binary string as an input. Consider the NOT gate, which flips any bit’s value. In qubits, this operation is called *x*. We begin by creating a new circuit dedicated to encoding, and we call it qc\_encode. For now, we only specify the number of qubits.

qc\_encode = QuantumCircuit(8)

qc\_encode.x(7)

qc\_encode.draw()

A screenshot of a white background

Description automatically generated

The code applies a not gate to our 8th qubit. Now we extract our results:

qc\_encode.measure\_all()

qc\_encode.draw()

A graph with black squares and arrows

Description automatically generated

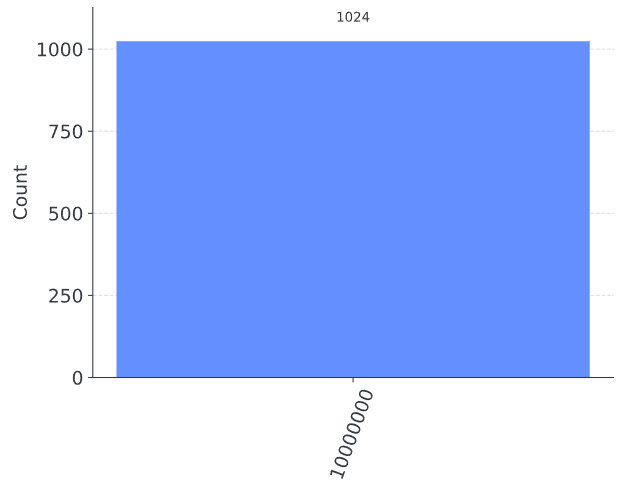
We can now run the circuit and view the results.

sim = Aer.get\_backend(‘aer\_simulator’)

result = sim.run(qc\_encode).result()

counts = result.get\_counts()

plot\_histogram(counts)



The computer now outputs 10000000 instead. The flipped bit lives on the far left of the string, since Qiskit numbers strings from right to left. This can be advantageous, since our 7th bit tells us how many 2^7s we have in our number. So by flipping the bit, we wrote the number 128.

We can encode any number in this way. For example, at the time of writing, I am 18. I can encode the number 18 through the code:

qc\_encode = QuantumCircuit(8)

qc\_encode.x(1)

qc\_encode.x(4)

qc\_encode.draw

A white background with black lines and green squares

Description automatically generated

To turn inputs into outputs, we need a problem to solve. We can solve the following math problem,

by breaking it down into more manageable parts. For example, . These are the same principles behind all addition algorithms, since computers need to be compiled to perform the simplest tasks possible. For example, take the above problem in binary.

The second number has a bunch of extra 0s on the left to make the two strings the same length. We add this number column by column, in the same manner as normal addition. However, in binary, the number 2 is written as 10, and requires 2 bits of information to store. Thus, for any value 2, we simply write a 0 and carry the 1. It follows that:

A *half adder* is a program that adds two values together. By chaining enough half adders together, a computer can calculate anything. Let’s make a half adder. This requires a circuit to encode the input, a circuit to executes the algorithm, and a circuit that encodes the output. The first part changes based on the desired input, but the rest remains the same.

A diagram of a diagram

Description automatically generated

The two bits we want to add are qubits 0 and 1. In the above example, they have been encoded to both have a value of 1, and so it seeks to solve 1 + 1. The result will be a string of two bits, which will be read from qubits 2 and 3 and stored in classical bits 0 and 1, respectively.

The basic operators of computers are called logic gates. We’ve already considered the NOT gate, but we need more powerful gates to have the computer perform addition. The half adder needs to be able to compute the following:

The rightmost bit in our answer is determined by whether the two bits we’re adding are equal or not. If they’re different, the rightmost bit is 1. If they’re the same, the rightmost bit is 0. An XOR gate is a logic gate that outputs true if one and only one input is true, and false if both or neither input is true. In quantum computing, the XOR gate’s job is done by the controlled-NOT gate, usually called CNOT, or cx in Qiskit. In circuit diagrams, it’s drawn as the following:

qc\_cnot = QuantumCircuit(2)

qc\_cnot.cx(0,1)

qc\_cnot.draw

A diagram of a connection

Description automatically generated with medium confidence

This operation applies to a pair of qubits. The small circle depicts the control qubit, and the large circle with the + depicts the target qubit. The CNOT gate reads the values of the control and target qubit, and overwrites the target qubit’s value with 0 if they’re the same, or 1 if they’re different.

A blue and black lines with a plus and a white cross

Description automatically generated

Another way to understand the CNOT gate is that it performs NOT on the target qubit if the control qubit is 1. For our half adder, we don’t want to overwrite one of our inputs. Instead, we want to write to an output. This can be done with 2 CNOTs.

qc\_ha = QuantumCircuit(4,2)

# encode inputs in qubits 0 and 1

qc\_ha.x(0) # For a=0, remove this line. For a=1, leave it.

qc\_ha.x(1) # For b=0, remove this line. For b=1, leave it.

qc\_ha.barrier()

# Use CNOTs to write the XOR of the inputs onto qubit 2

qc\_ha.cx(0,2)

qc\_ha.x(1,2)

qc\_ha.barrier()

qc\_ha.measure(2,0) # extract XOR value into classical bit 0

qc\_ha.measure(3,1)

qc\_ha.draw()

A diagram of a diagram

Description automatically generated

If qubit 0 is a 0, CNOT will output false to qubit 2. If qubit 0 is a 1, CNOT will output true to qubit 2. Thus, the first CNOT just copies the initial input to qubit 2. The second CNOT overwrites qubit 2 with the rightmost digit of the addition, as we saw earlier.

We now need to finish the adder by encoding qubit 3, wherein the second digit of the addition sequence will live.

The only case for which the leftmost digit can be a 1 is if both q0 and q1 are 1s. To calculate this, we can look at qubits 0 and 1, and if they both are 1, we perform NOT on qubit 3. For this, we need a new logic gate, similar to a CNOT gate but with 2 controls instead of 1. This is the Toffoli gate, which is similar to the AND gate in classical computing. In Qiskit, the Toffoli gate is represented with ccx.

qc\_ha = QuantumCircuit(4,2) # 4 qubits, 2 classical bits for storage

# encode inputs in qubits 0 and 1

qc\_ha.x(0) # for a=0, remove this line. For a=1, keep it.

qc\_ha.x(1) # for b=0, remove this line. For b=1, keep it.

qc\_ha.barrier()

# use cnots to write the XOR of the inputs on qubit 2

qc\_ha.cn(0,2)

qc\_ha.cn(1,2)

# use ccx to write the AND of the inputs on qubit 3

qc\_ha.ccx(0,1,3)

qc\_ha.barrier()

# extract outputs

qc\_ha.measure(2,0)

qc\_ha.measure(3,1)

qc\_ha.draw()

A diagram of a circuit

Description automatically generated

In this example, we are calculating 1+1 since both inputs are 1. Let’s view the result.

qobj = assemble(qc\_ha)

counts = sim.run(qobj).result().get\_counts()

plot\_histogram(counts)

A blue rectangular bar graph

Description automatically generated

The result is 10, which is the binary equivalent of 2. We have built a computer that can solve 1+1. The inputs can be modified, but the computer will still output the proper result.

The NOT, CNOT, and Toffoli gates can create programs that add numbers of any size. These three gates are all that’s really needed in quantum computing. In fact, you can do without the CNOT gate as well. The NOT gate is only really needed to create bits with value 1. The Toffoli gate is the most basic building block of Quantum Computing.

**Representing Qubit States**

Complex variables, data structures, and objects are really just big piles of classical bits, which we call classical variables. The computers that use them are called classical computers. In quantum computing, the basic variable is the qubit, which is a variation of the bit. They can only ever store a single piece of binary information, but are governed by quantum mechanics, and can thus be subject to different gates and can be used to design different algorithms. To write down qubit states, we use the mathematics of matrices, vectors, and complex numbers.

In quantum physics we use the *statevector* to describe a quantum system. Consider a car at some position on a track, described as . An alternative method of describing the car’s position is to have a statevector, which contains the probabilities of finding the car at any certain place:

This isn’t limited to position, as a statevector can describe any quantity. In classical physics, this is a silly thing to do, since it requires giant vectors to store values that can be described as a single number. However, this is an exceptional method of storing quantum information.

Classical bits (c) are only ever 0 or 1. To write down a bit’s state, we can write something like:

For quantum bits, this restriction doesn’t apply. Whether we get a 0 or 1 only needs to be well defined when measurement is done to extract an output, but otherwise its state can be more complex than a classical bit. The qubit must have two mutually exclusive states, wherein the 0 state corresponds to an output of 0, and the 1 state corresponds to an output of 1. This can be represented mathematically by the two orthogonal vectors:

With vectors, we can describe more complicated states than just 0 and 1. For example, take the following vector:

To understand this vector, we need to be able to multiply it by scalar quantities. The states and form an orthonormal basis, orthonormal meaning both orthogonal and normal (normal means their magnitudes = 1). This allows us to describe any 2d vector as a combination of theses states. For example,

The vector is the qubit’s state vector, and it tells us everything we can know about the qubit. So far, we can only conclude that is a linear combination of the two basis vectors. In quantum mechanics, we describe linear combinations such as this with the word superposition. Our state is just as well defined as the basis vectors and can be manipulated as such.

Let’s examine how Qiskit allows us to manipulate qubits. First, we import all the tools we need.

from qiskit import QuantumCircuit, assemble, Aer

from qiskit.visualization import plot\_histogram, plot\_bloch\_vector

from math import sqrt, pi

In Qiskit, we use QuantumCircuit to store our circuits. It’s essentially a list of the quantum operations on our circuit and the qubits they’re applied to.

qc = QuantumCircuit(1) # creates a quantum circuit with 1 qubit

Our qubits always start in the state . We can use the initialize() method to transform this into any state. We give initialize() the vector we want in the form of a list, and tell it what qubit we want to apply it to.

qc = QuantumCircuit(1) # creates a quantum circuit with 1 qubit

initial\_state = [0,1] # define initial\_state as |1>

qc.initialize(initial\_state, 0) # initialize 0th qubit

qc.draw() # view the circuit

A purple square with white text

Description automatically generated

We then use one of Qiskit’s simulators to view the resulting state of the qubit.

sim = Aer.get\_backend(‘aer\_simulator’) # tell Qiskit how to simulate

To get the results from our circuit, we use run to execute our circuit, giving the circuit and the backend as arguments. We then use .result() to get the result of this:

qc = QuantumCircuit(1) # creates a quantum circuit with 1 qubit

initial\_state = [0,1] # define initial state as |1>

qc.initialize(initial\_state, 0) #initialize 0th qubit

qc.save\_statevector() # tell simulator to save statevector

qobj = assemble(qc) # create Qobj from the circuit for the sim to run

result = sim.run(qobj).result() # do the simulation and return result

from result, we can then get the final statevector using .get\_statevector():

out\_state = result.get\_statevector()

print(out\_state) # display the output state vector



Python uses j to represent i in complex numbers. We thus read the output statevector as:

Let’s now measure our qubit as we would in a real quantum computer and see the result:

qc.measure\_all()

qc.draw()

A diagram of a statevector

Description automatically generated

This time, instead of a statevector we will get the counts for 0 and 1 results using .get\_counts():

qobj = assemble(qc)

result = sim.run(qobj).result()

counts = result.get\_counts()

plot\_histogram(counts)

A blue rectangular object with white text

Description automatically generated

We find a 100% chance of measuring , which make sense since we initialized our qubit as , and did not perform any more manipulations to it.

Now, let’s consider a superposition. Let’s use our state from earlier:

We simply change the amplitudes to our list. Python uses j for the imaginary unit, instead of i.

initial\_state = [1/sqrt(2), 1j/sqrt(2)] # sets initial\_state to |q0>

We now re-initialize the qubit:

qc = QuantumCircuit(1) # we have to redefine qc

qc.intialize(initial\_state, 0) # initalize to |q0>

qc.save\_statevector # save statevector

qobj = assemble(qc)

state = sim.run(qobj).result().get\_statevector() # sim the circuit

print(state) # prints the result



qobj = assemble(qc)

results = sim.run(qobj).results.get\_counts()

plot\_histogram(results)

A graph with blue rectangular bars

Description automatically generated with medium confidence

We see an equal probability of measuring or .

There is a very simple rule for measurement. For any qubit of state , the probability of measuring it in a state is such that:

where . If we consider our state , we see that the probability of measuring is indeed 0.5:

As practice, let’s verify the probability of measuring is indeed the same.

This rule governs how we get information out of quantum states and is very important for all of quantum computing. It also implies several facts.

#1 Normalization

The rule shows us that amplitudes are related to probabilities. If we want the probabilities to sum to 1, which they need to, the statevector must be properly normalized. Specifically, we require the magnitude of the statevector to be 1, meaning:

Thus if:

for some quantum state .

This explains the factors of . Qiskit gives errors for initialization parameters that don’t satisfy normalization.

Exercise:

Create a state vector that will give a probability of measuring :

Verify that the probability of measuring for this state is :

#2 Alternative Measurement

The measurement rule gives us that a state is measured as , but it does not specify to be or . For any orthogonal pair of states, we can define a measurement that would cause a qubit to choose between the two. This will be explored more later.

#3 Global Phase

We know the state gives us 1 with certainty. However, we can write states such as . We consider the behavior of these states via the measurement rule:

We find that the factor disappears when we take the magnitude of the complex number. This effect is independent of the state . Since measurement is the only way to extract information from a qubit, these states are considered equivalent in all ways that are physically relevant.

More generally, we refer to any value for which as a **global phase**. States that differ only by a global phase are indistinguishable.

This is distinct from the phase difference between terms in a superposition, which is known as **relative phase**. This becomes relevant once we consider different types of measurement and multiple qubits.

#4 The Observer Effect

The amplitudes contain information about the probability of us finding the qubit in any specific state, however once we measure a certain state, subsequent measurements will return the same state for the qubit.

This is often referred to as collapsing the state of the qubit. This is a potent effect and must be used wisely. This disallows continuous measurement of a qubit in a circuit, otherwise it would always be in a well-defined state. Thus, measurements are only used when we need to extract an output.

This can be demonstrated with Qiskit’s simulator functions. We initialize a qubit in a superposition:

qc = QuantumCircuit(1)

initial\_state = [1j/sqrt(2), 1/sqrt(2)]

qc.initialize(initial\_state, 0)

qc.draw()

A purple rectangular object with black text

Description automatically generated

This initializes our qubit in the state:

Now, we create a circuit in which the qubit is measured:

qc = QuantumCircuit(1)

initial\_state = [1j/sqrt(2), 1/sqrt(2)]

qc.initialize(initial\_state, 0)

qc.measure\_all()

qc.save\_statevector()

qc.draw()

When we simulate this entire circuit, we see that one of the amplitudes is always 0:

qobj = assemble(qc)

state = sim.run(qobj).result().get\_statevector()

print(“State of Measured Qubit = “ + str(state))

Any measured values are all or . Although either outcome is equally probable, the qubit is never actually in a superposition since it’s already been measured. If it was, the statevector would give a nonzero amplitude to both basis vectors.

These are just simulations. In a real quantum computer, we can only ever receive a 1 or 0 from it. The output of a 10-bit quantum computer would look something like:

1001011011

Notice there are no complex or fractional values. Qiskit provides multiple simulators that allow us to look at the state of a qubit during operation, but doing this on a real quantum computer would collapse the superposition.

The general state of a qubit is:

The first two implications of the measurement rule tell us we cannot differentiate between some of these states. Thus, we can be more specific in the description of a qubit. Since we cannot measure global phase, only the differences in phase between the basis vectors, we can define alpha and beta as reals, and add a complex term to tell us the relative phase between them:

Since the qubit state must be normalized, it follows that:

We can use the trigonometric identity:

to describe the real and in terms of one variable:

We use this fact to describe the qubit’s state in terms of two variables:

We can interpret and as spherical coordinates, with since the magnitude of the qubit state is always 1, and thus can plot any qubit state on the surface of a sphere, known as the Bloch sphere. Consider the state , where and :

A sphere with a red line

Description automatically generated

Remember: the Bloch vector is a visualization tool that maps the 2D statevector into 3D space.

**Single Qubit Gates**

**The Pauli Gates**

The Pauli-X gate is represented by the Pauli-X matrix:

To see the effect the gate has on a qubit, we simply multiply the qubit’s statevector by the gate:

We can see that the Pauli-X gate switches the amplitude of the states and . The Pauli-X gate is sometimes referred to as the NOT gate, referencing its classical analogue. It can be thought of as a rotation of radians about the x axis of the Bloch sphere.

The Pauli-Y gate is represented by the Pauli-Y matrix:

The Pauli-Y gate can be thought of as a rotation of radians about the y axis of the Bloch sphere.

The Pauli-Z gate is represented by the Pauli-Z matrix:

The Pauli-Z gate has no effect on the two basis states. This is because these are the eigenstates (eigenvalue corresponding to the wave equation of a quantum state) of the Pauli-Z matrix. In fact, the computational basis (the basis formed by the states and ) is often called the Z-basis. To form a basis, you simply need two orthogonal vectors, so there are actually infinitely many bases that can be used to construct the vector space. Sometimes the eigenstates of the Pauli-X gate are used to construct the X-basis. These are as follows:

Another less commonly used basis is the eigenstates formed by the Pauli-Y gate. These are called:

Using only the Pauli gates, it’s not possible to achieve superposition. We can only achieve states and .

Exercise: find the eigenstates of the Pauli-Y gate:

Suppose

Suppose

The eigenvector is actually satisfied for any . I believe these are the basis vectors, however. They appear to be linearly independent.

I was incorrect, and I’m not quite sure why.

I understand why – the vectors are not normalized, and thus the resultant Hilbert space doesn’t describe a valid quantum system. It follows that:

Correct!

Exercise: find the coordinates of the basis vectors of the Y-Basis on the Bloch sphere.

**The Hadamard Gate**

The Hadamard gate (H-gate) is a fundamental quantum gate that allows us to create superposition. It has the matrix:

It performs the following transformations:

Exercises:

Write the H-gate as the outer products of and .

Etc.

Show that applying a sequence of gates: HZH, to any qubit state is equivalent to applying an X-gate.

Find a combination of X, Z, and H-gates that is equivalent to a Y-gate, ignoring global phase.

Find the matrix:

HZXH is equivalent to a Y-gate, ignoring global phase.

We have seen that the Z-axis is not special, and in fact there are infinitely more bases. We don’t always have to measure our qubits in the Z-basis, we can in fact measure in any base we want. For example, let’s measure in the X-basis. We calculate the probability of measuring each basis vector as:

After measurement, superposition is destroyed. Qiskit only allows for measurement in the Z-basis, so we have to create our own with Hadamard gates:

# Create the X-measurement function:  
def x\_measurement(qc, qubit, cbit):

# Measure ‘qubit’ in the X-basis, store the result in cbit

qc.h(qubit)

qc.measure(qubit, cbit)

return qc

initial\_state = [1/sqrt(2), -1/sqrt(2)]

# Initialize our qubit and measure it

qc = QuantumCircuit(1,1)

qc.initialize(initial\_state,0)

x\_measurement(qc, 0, 0)

qc.draw()

A diagram of a chemistry experiment

Description automatically generated

We saw the gates HZH act as an X gate. This is since the H-gate switches our qubit to the X-basis, the Z-gate performs a NOT in the X-basis, and the H-gate switches the qubit back to the Z-basis. Following the same logic, the above measurement is in the X-basis since we have transformed the X-basis into the Z-basis before our measurement.

There’s another way to see how the Hadamard gate takes us to the X-basis. Suppose we wish to measure the X-basis in the normalized state . To measure it in the X-basis, we must express it in terms of and . Notice the following relations:

Observe the probability amplitudes in X-basis can be obtained by applying a Hadamard matrix on the statevector expressed in Z-basis.

We now consider the results of the earlier X-measurement program:

qobj = assemble(qc) # assemble circuit into a Qobj that can be run

counts = sim.run(qobj).result().get\_counts() # do the simulation

plot\_histogram(counts) # display the output on measure of statevector

A blue rectangular object with white text

Description automatically generated

We initialized our qubit in the state , but we can see that, after measurement, we have collapsed our qubit to the state . This is since along the X-basis, is a base state and measuring it along the X-basis will always yield the same result.

This allows us to show the Heisenberg uncertainty principle. If the qubit is initialized in the state , our measurement in the Z-basis is certain to be , but our measurement in the X basis is fully random. Likewise, for a qubit initialized as , our X-basis measurement is guaranteed to be , but our Z-basis measurement is fully random. More generally: whatever state a quantum system is in, there’s always a measurement that has a deterministic outcome.

**The P-Gate**

The P-gate, or phase-gate, is *parametrized*. In other words, it requires a value in order to specify what it does. It performs a rotation about the Z-axis. In matrix form,

For . In Qiskit, we specify a P-gate using p(phi, qubit):

qc = QuantumCircuit(1)

qc.p(pi/4, 0)

qc.draw()

A purple square with black text

Description automatically generated

Notice that the P-gate is a generalization of the Z-gate, as:

In fact, the next three gates that will be referenced are all special cases of the P-gate.

**The I, S, and T-Gates**

The I-gate is the identity gate. It simply does nothing. Its matrix is the identity matrix:

This gate has no effect on qubits, so it’s interesting that it’s even considered a gate. It’s mainly used in calculations, for example proving X is its own inverse:

The other reason it’s used is when considering real hardware to specify a “do nothing” operation.

Exercise: what are the eigenstates of the I-gate?

For the basis state

Any nonzero vector is an eigenvector for the identity matrix. The basis vectors are .

The S-gate, also known as the -gate, is a P-gate with . It does a quarter turn about the Bloch sphere. The S-gate is the first gate that has been introduced so far that is not its own inverse. It’s also not Hermitian. Thus, you will see the following used often:

The name -gate comes from the fact that 2 successively applied S gates acts as one Z-gate:

In Qiskit:

qc = QuantumCircuit(1)

qc.s(0) # apply S-gate to qubit 0

qc.sdg(0) # apply S-dagger to qubit 0

qc.draw

A blue square with black text

Description automatically generated

The T-gate is a P-gate with . It is also not Hermitian or Unitary.

As with the S-gate, the T-gate is sometimes known as the -gate. In Qiskit:

qc = QuantumCircuit(1)

qc.t(0) # apply T-gate to qubit 0

qc.tdg(0) # apply T-dagger to qubit 0

qc.draw()

A purple squares with black letters

Description automatically generated

**The U-gate**

The U-gate is a generalized form of all quantum gates, similarly to how the P-gate is a generalized form of the Z, T, S, and I-gates. It’s of the form:

Every gate given herein can be specified of the form , however this is rarely seen in circuit diagrams due to the difficulty of reading.

As an example, lets consider the H-gate and P-gates:

Consider the Qiskit code to transform a state to a state.

qc = QuantumCircuit(1)

qc.u(pi\2, 0, pi, 0)

qc.draw()

A purple square with black text

Description automatically generated

We view the result:

qc.save\_statevector()

qobj = assemble(qc)

state = sim.run(qobj).result().get\_statevector()

plot\_bloch\_statevector(state)

A sphere with lines and a red arrow

Description automatically generated

It follows that there must be an infinite number of gates. It should also be noted that there is nothing special about the Z-basis, except that it has been selected as the standard basis for computing. Qiskit provides X-basis equivalents of the S and Sdg gates, SX and SXdg respectively. These gates do a quarter turn about the x axis.

Before running on real IBM quantum hardware, all single-qubit operations are compiled down to and . For this reason these are sometimes known as the physical gates.