CPEN 400Q / EECE 571Q Lecture 02
Quantum circuits and PennyLane

Thursday 13 January 2022

Announcements

- Classes are online until 7 Feb
- Piazza has been setup for the class
- Assignment 0 due on Tuesday before class
  - Instructions have been updated
  - Submit GitHub username/student ID as text response
  - Please update forked repo permissions
  - Make PR to master branch on *your* copy of the repo
- Assignment 1 will be available tomorrow (due in 2 weeks; lots of time)

Last time

We learned that qubits are physical systems whose states are represented by complex-valued vectors that are linear combinations of two **basis states** $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad |\alpha|^2 + |\beta|^2 = 1.$$

Last time

A qubit lives in a 2-dimensional complex vector space with an **inner product** called a **Hilbert space**. The inner product tells us about the *overlap* between two states.

Last time

The coefficients in the linear combination (**amplitudes**) tell us the probability of observing a particular basis state $|\psi_i\rangle$ when we **measure** a qubit.

We can compute these probabilities by projecting onto basis states using the inner product.

$$\text{Pr(outcome i)} = |\langle\psi_i|\varphi\rangle|^2$$

In between state preparation and measurement, we apply 2×2 **unitary matrices** (gates/operations) to modify the qubit's state.

A matrix U is unitary if

$$UU^\dagger = U^\dagger U = \mathbb{1}.$$

Unitaries preserve the length of qubit state vectors, and the angles between them.

Last time

We wrote some NumPy code to do all this:

```
def ket_0():
    return np.array([1, 0])

def ket_1():
    return np.array([0, 1])


def superposition(alpha, beta):
    return alpha * ket_() + beta * ket_1()


def apply_op(U, state):
    return np.dot(U, state)

def apply_ops(list_U, state):
    for U in list_U:
        state = np.dot(U, state)
    return state
```

Last time

```python
def measure(state, num_samples):
    # Compute using the inner product method
    prob_0 = np.abs(np.vdot(ket_0(), state)) ** 2
    prob_1 = np.abs(np.vdot(ket_1(), state)) ** 2

    samples = np.random.choice(
        [0, 1], size=num_samples, p=[prob_0, prob_1]
    )

    return samples
```

Last time

Quantum computing involves preparing a qubit in a particular state, applying one or more unitary operations, and performing a measurement.

```
def quantum_algorithm(alpha, beta, list_U):
    initial_state = superposition(alpha, beta)
    state = apply_ops(initial_state, list_U)
    return measure(state)
```

But doing all of this both by hand or using pure NumPy can be tedious, so today we will shift from NumPy to the quantum software framework PennyLane.

Learning outcomes

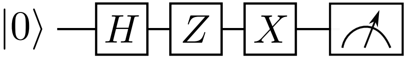
- Implement single-qubit quantum algorithms in PennyLane
- Describe the behaviour of common single-qubit gates
- Calculate the expectation value of an observable
- Perform measurements in other bases

Recall three of our quantum gates from last time:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

We can apply these gates to a qubit and express the computation in matrix form, or as a quantum circuit.

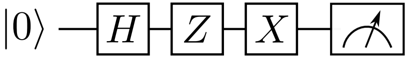
$$XZH|0\rangle$$



Quantum functions

We can also express this circuit as a **quantum function** in PennyLane.

$$XZH|0\rangle$$




```python
import pennylane as qml

def my_quantum_function():
    qml.Hadamard(wires=0)
    qml.PauliZ(wires=0)
    qml.PauliX(wires=0)
    return qml.sample()
```

Quantum functions

Quantum functions are like normal Python functions, with two special properties:

1. Apply one or more quantum operations

```
import pennylane as qml

def my_quantum_function():
    qml.Hadamard(wires=0) # Apply Hadamard gate to qubit 0
    qml.PauliZ(wires=0)   # Apply Pauli Z gate to qubit 0
    qml.PauliX(wires=0)   # Apply Pauli X gate to qubit 0
    return qml.sample()
```

Q: Why `wires`? A: PennyLane can be used for continuous-variable quantum computing, which does not use qubits.

Quantum functions

Quantum functions are like normal Python functions, with two special properties:

1. Apply one or more quantum operations
2. Return a measurement on one or more qubits

```
import pennylane as qml

def my_quantum_function():
    qml.Hadamard(wires=0)
    qml.PauliZ(wires=0)
    qml.PauliX(wires=0)
    return qml.sample() # Return measurement samples
```

Quantum functions are executed on **devices**. These can be either *simulators*, or *actual quantum hardware*.

```
import pennylane as qml

dev = qml.device('default.qubit', wires=1, shots=100)
```

This creates a device of type `'default.qubit'` with 1 qubit that returns 100 measurement samples for anything that is executed.

Quantum functions

A **QNode (quantum node)** is an object that binds a quantum function to a device, and executes it.

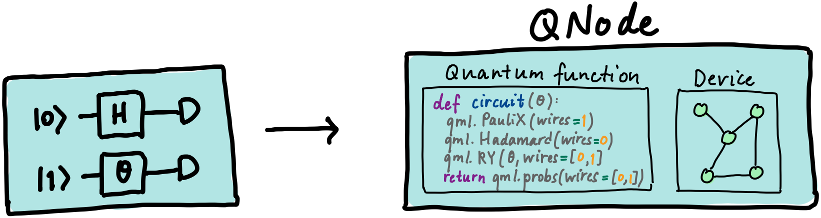


Image credit: `https://pennylane.ai/qml/glossary/quantum_node.html`

Quantum nodes

```python
import pennylane as qml

dev = qml.device('default.qubit', wires=1, shots=100)

def my_quantum_function():
    qml.Hadamard(wires=0)
    qml.PauliZ(wires=0)
    qml.PauliX(wires=0)
    return qml.sample()
```

With these two components, we can create and execute a QNode.

```python
# Create a QNode
my_qnode = qml.QNode(my_quantum_function, dev)

# Execute the QNode
result = my_qnode()
```

Hands-on with QNodes

You probably have some questions...

1. Where's the state?

You probably have some questions...

1. Where's the state?
   - Inside the device!

You probably have some questions...

1. Where's the state?
   - Inside the device!
2. What happens to the gates?

You probably have some questions...

1. Where's the state?
   - Inside the device!
2. What happens to the gates?
   - Operations are recorded onto a "tape"

You probably have some questions...

1. Where's the state?
   - Inside the device!
2. What happens to the gates?
   - Operations are recorded onto a "tape"
   - The QNode constructs the tape when it is called

You probably have some questions...

1. Where's the state?
   - Inside the device!
2. What happens to the gates?
   - Operations are recorded onto a "tape"
   - The QNode constructs the tape when it is called
   - The tape is then executed on the device.

Single-qubit unitary operations

More quantum gates

So far, we know 3 gates that do the following:

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

But a general qubit state looks like

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

where α and β are *complex numbers* (such that $|\alpha|^2 + |\beta|^2 = 1$).

How do we make the rest?

Consider the operation Z:

$$Z|0\rangle = |0\rangle, \quad Z|1\rangle = -|1\rangle.$$

Apply this to a superposition:

The *sign* of the amplitude on the $|1\rangle$ state has changed.

Z rotations

We know that $-1 = e^{i\pi}$:

What if instead of π, we used an arbitrary angular parameter?

The extra $e^{i\theta}$ is called a **relative phase**.

The "proper" form of this rotation is

$$RZ(\theta) = e^{-i\frac{\theta}{2}Z} = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$$

Exercise: expand out the exponential of Z to obtain the matrix representation.

S and T

Two other special cases: $\theta = \pi/2$, and $\theta = \pi/4$.

$$\begin{aligned} S &= RZ(\pi/2) = \begin{pmatrix} e^{-i\frac{\pi}{4}} & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix} \sim \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \\ T &= RZ(\pi/4) = \begin{pmatrix} e^{-i\frac{\pi}{8}} & 0 \\ 0 & e^{i\frac{\pi}{8}} \end{pmatrix} \sim \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix} \end{aligned}$$

S is part of a special group called the **Clifford group**.

T is used in universal gate sets for fault-tolerant QC.

X and Y rotations

RZ changes the phase, but not the magnitudes of the amplitudes.
How do we change those?

RX, and RY rotations...

There is a reason we are calling these rotations.

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

We can rewrite $\alpha = ae^{i\phi}$ and $\beta = be^{i\omega}$ where a, b are real-valued numbers:

Factor out the $e^{i\phi}$ (a **global phase**):

“Rotations”?

The global phase doesn’t matter though!

It does not affect the measurement outcome probabilities.

“Rotations”?

If the global phase doesn't matter...

$$|\psi\rangle = e^{i\phi}\left(a|0\rangle + be^{i(\omega-\phi)}|1\rangle\right) \sim a|0\rangle + be^{i(\omega-\phi)}|1\rangle$$

Relabel $\varphi = \omega - \phi$:

$$|\psi\rangle = a|0\rangle + be^{i\varphi}|1\rangle$$

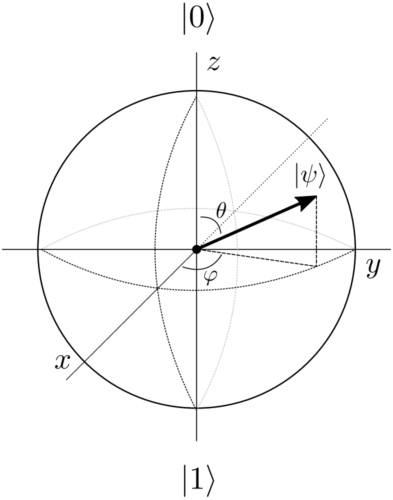
“Rotations”?

Normalization tells us that $a^2 + b^2 = 1$. What else has this relationship?

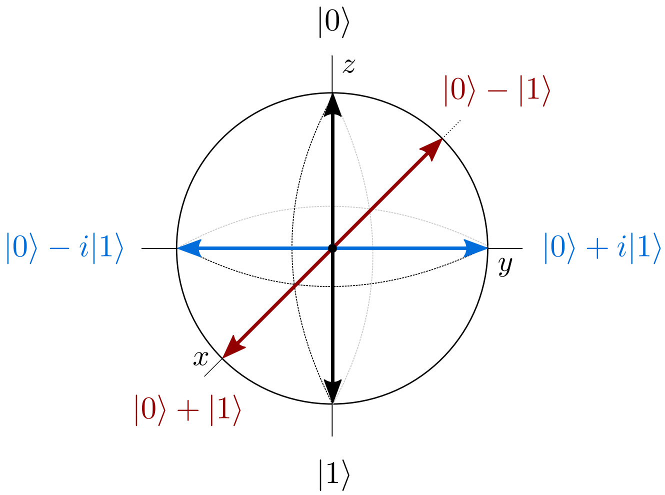
We can rewrite as:

So any single-qubit state can be specified by two angular parameters... just like points on a sphere!

Rotations: the Bloch sphere



Rotations: the Bloch sphere



Rotations: the Bloch sphere

RX, RY, and RZ correspond visually to rotations about their respective axes.

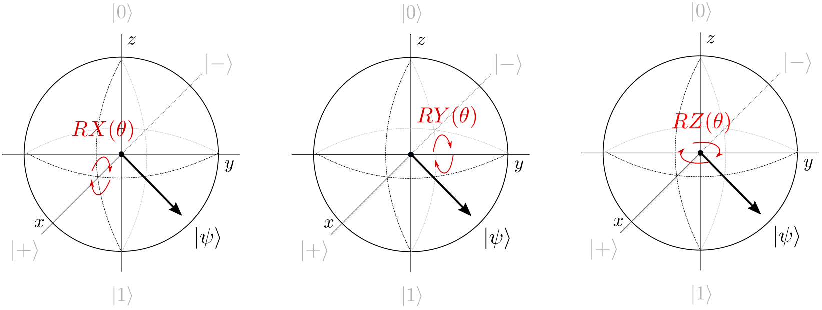
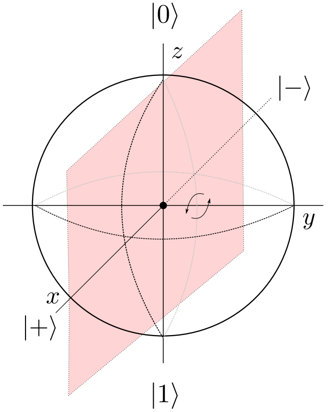
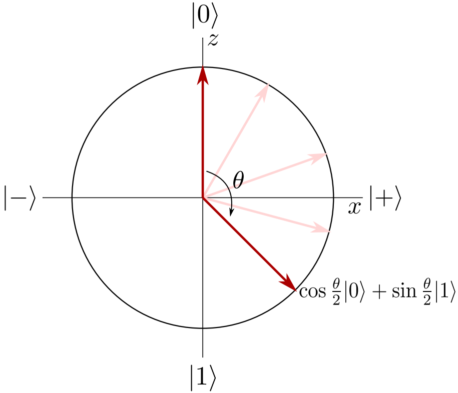


Image credit: Codebook node I.6

Rotations: RY

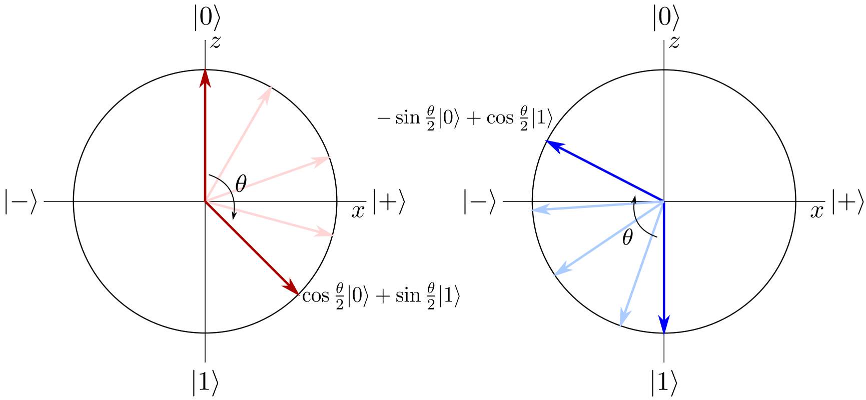




Rotations: *RY*

The matrix representation of *RY* is

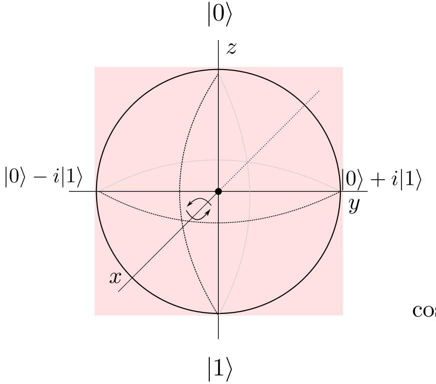
$$RY(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

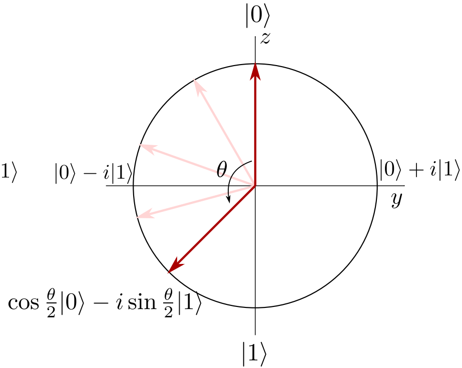


Rotations: RX

RX is similar but has complex components:

$$RX(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$





Pauli rotations

These unitary operations are called **Pauli rotations**.

	Math	Matrix	Code	Special cases
RZ	$e^{-i\frac{\theta}{2}Z}$	$\begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$	`qml.RZ`	$Z(\pi), S(\pi/2), T(\pi/4)$
RY	$e^{-i\frac{\theta}{2}Y}$	$\begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$	`qml.RY`	$Y(\pi)$
RX	$e^{-i\frac{\theta}{2}X}$	$\begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$	`qml.RX`	$X(\pi), SX(\pi/2)$

Adjoints

We can rotate forwards, or backwards by negating the angle. But there is a more general way of rotating backwards. In PennyLane, we can compute adjoints of operations *and* entire quantum functions using `qml.adjoint`:

```
def some_function(x):
    qml.RZ(Z, wires=0)

def apply_adjoint(x):
    qml.adjoint(qml.S)(wires=0)
    qml.adjoint(some_function)(x)
```

`qml.adjoint` is a special type of function called a **transform**. We will cover transforms in more detail around beginning of week 4.

Hands-on time...

What about H?

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

This does not have the form of RX, RY, or RZ.

But, we can use a combination of these to make an H (actually, just need two of the three).

Deep dive: unitary operations

The $n \times n$ unitary matrices are a mathematical group under matrix multiplication, $U(n)$:

1. Closure: for U, V unitary, UV is also unitary
2. Associativity: $(UV)W = U(VW)$
3. Identity: $\mathbb{1}$
4. Inverses: $U^{-1} = U^\dagger$

Deep dive: unitary operations

The $n \times n$ unitary matrices are a mathematical group under matrix multiplication, $U(n)$:

1. Closure: for U, V unitary, UV is also unitary
2. Associativity: $(UV)W = U(VW)$
3. Identity: $\mathbb{1}$
4. Inverses: $U^{-1} = U^\dagger$

Any unitary matrix can be written in terms of a finite set of real-valued parameters:

$$U(\phi, \theta, \omega) = e^{i\alpha} \begin{pmatrix} e^{-i(\phi+\omega)/2}\cos(\theta/2) & -e^{i(\phi-\omega)/2}\sin(\theta/2) \\ e^{-i(\phi-\omega)/2}\sin(\theta/2) & e^{i(\phi+\omega)/2}\cos(\theta/2) \end{pmatrix}$$

With just RZ and RY (or RZ/RX, RY/RX), we can implement *any single-qubit unitary operation*[1]:

$$U = e^{i\alpha} RZ(\omega) RY(\theta) RZ(\phi)$$

$\{RZ, RY\}$ is **universal** for single-qubit quantum computing.

Hands-on...

For more fun: do text exercises in Codebook node I.3 and I.7.

[1] Note that the α technically doesn't matter.

With just H and T, we can approximate any single-qubit rotation up to arbitrary accuracy. For example, we can implement $RZ(0.1)$ up to accuracy 10^{-10}:

```
→ gridsynth 0.1 -d 10
HTHTHTHTHTSHTHTHTHTSHTSHTHTHTSHTSHTHTSHTHTSHTHTHTSHTSHTHTSHTSHTSHTS
HTHTHTHTHTHTHTHTHTHTHTSHTSHTSHTSHTSHTSHTHTSHTSHTSHTSHTHTHTSHTSHTHT
SHTSHTSHTHTHTHTSHTHTHTSHTSHTHTHTHTHTSHTHTHTSHTSHTSHTSHTSHTHTSHTHTHT
HTHTHTHTHTSHTHTHTSHTHTSHTSHTHTHTSHTSHTHTSHTSHTHXWWW
```

This was generated using the `newsynth` Haskell package:
`https://www.mathstat.dal.ca/~selinger/newsynth/`

Universal gate sets: H and T

Or to accuracy 10^{-100}:

```
→ gridsynth 0.1 -d 100
SHTSHTHTSHTSHTHTSHTHTSHTSHTHTSHTHTSHTSHTSHTHTSHTHTSHTSHTSHTHTHTSHTHTSHTHTHTHTS
HTSHTHTSHTSHTSHTSHTSHTHTHTHTSHTHTHTHTHTSHTHTSHTSHTHTHTSHTHTSHTHTHTSHTHTSHTSHTS
HTHTSHTHTHTSHTSHTSHTSHTSHTSHTHTHTSHTSHTSHTHTHTSHTHTHTHTHTSHTHTSHTHTSHTHTSHTSHT
SHTSHTHTSHTHTSHTSHTHTHTSHTHTHTSHTSHTHTHTHTHTHTHTHTHTHTHTHTHTHTSHTSHTSHTSHTHTHT
SHTSHTSHTHTSHTHTHTSHTHTSHTSHTSHTHTHTHTHTHTHTSHTSHTSHTSHTHTHTHTSHTSHTSHTSHTSHTH
THTHTSHTSHTSHTSHTSHTHTHTHTHTSHTHTHTSHTHTSHTHTHTSHTHTHTSHTHTSHTHTSHTSHTSHTSHTHT
HTHTSHTSHTSHTHTSHTHTSHTSHTHTHTHTHTHTHTSHTHTHTHTHTSHTHTHTHTSHTSHTSHTSHTSHTSHTSH
TSHTSHTSHTSHTSHTSHTSHTHTSHTHTSHTSHTSHTHTSHTSHTSHTSHTHTHTSHTHTHTHTHTHTHTHTHTHTS
HTHTHTHTHTSHTSHTHTHTSHTHTSHTSHTSHTSHTHTHTSHTSHTSHTHTHTSHTHTHTHTHTSHTSHTSHTSHTS
HTHTSHTSHTHTHTHTHTHTHTHTSHTSHTHTSHTSHTHTHTSHTSHTHTSHTSHTSHTSHTSHTSHTHTHTSHTHTS
HTSHTHTHTHTHTSHTSHTSHTSHTSHTSHTSHTSHTHTSHTSHTHTHTSHTSHTHTHTSHTSHTSHTHTSHTHTSHT
HTHTHTSHTHTSHTHTSHTSHTHTSHTHTSHTHTHTSHTSHTHTSHTHTHTHTHTHTHTSHTHTSHTHTHTSHTHTSH
TSHTHTSHTHTHTSHTHTSHTSHTHTSHTSHTHTSHTHTSHTSHTSHTSHTHTSHTHTSHTHTSHTSHTSHTSHTHTH
THTHTSHTHTSHTHTSHTHTSHTHTSHTSHTHTHTHTSHTSHTHTHTHTSHTHTSHTSHTHTHTSHTHTSHTSHTSHT
SHTHTSHTSHTSHTSHTHTHTSHTHTHTHTSHTHTHTSHTHTSHTHTHTSHTSHTSHTSHTSHTSHTSHTHTHTHTSH
TSHTSHTSHTHTSHTSHTSHTHTHTSHTSHTHTSHTHTSHTSHTSHTHTSHTHTHTHTSHTSHTHTHTHTHTSHTHTH
TSHTHTSHTSHTSHTHTSHTHTSHTHTHTHTSHTHTSHTHTSHTHTSHTHTSHTHTHTHTSHTSHTSHTHTHTSHTHT
SHTHTHTSHTSHTSHTSHTHTSHTSHTHTHTHTSHTSHTHTSHTSHTSHTSHTHTHTSHTHTHTSHTSHTHTHTSHTH
TSHTHTHTSHTSHTHTHTSHTHTSHTHTHTHTHTSHTSHTSHTHTSHTSHTSHTSHTSHTSHTSHTSHTHTHTHTSHT
SHTHTHTSHTSHTSHTSHTSHTSHTSHTSHTSHTHTSHTHTSHTHTHTSHTHTSHTSHTSHTHTHTSHTSHTHTH
THTSHTHTSHTHTHTSHTSHTHTSHTHTSHTSHTSHTHTHTHTSHTSHTHTHTHTSHTHTSHTSHTSHTHTHTHTHTH
THTSHTSHTHTHTHTHTSHTSHTHTHTSHTSHTSHTHTHTSHTHTHTSHTHTSHTSHTHTHTSHTSHTHTSHTSHTSH
THTHTSHTHTHTHTHTSHTHTHTSHTSHTHTHTSHTHTHTSHTSHTSHTHTSHTSHTHTHTSHTSHTHTSHTSHTHTH
TSHTHTSHTSHTHTSHTHTSHTSHTHTSHTHTSHTHTHTSHTSHTSHTSHTSHTSHTSHTHTSHTHTSHTSHTHTHTS
HTHTSHTHTSHTHTHTSHTSHTSHTHTHTSHTHTSHTHTSHTHTSHTSHTSHTSHTSHTHTHTSHTSHTSHTSHTHTS
HTHTHTHTHTHTHTHTSHTHTHTHTSHTHTHTHTHTSHTSHTSHTHTHTSHTHTSHTHTSHTSHTSHTHTSHTHTSHT
HTHTHTSHTSHTSHTHTSHTSHTHTSHTSHTSHTSHTHTHTSHTSHTSHTHTHTSHTHTSHTSHTHTSHTSHTHTSHT
SHTSHTSHTHTHTSHTHTSHTHTSHTSHTHTSHTHTHTHTSHTSHTSHTSHTSHTSHTSHTHTSHTSHTHTHTSHTSH
THTHTHTHTHTHTHTSHTSHTSHTSHTHTSHTSHTSHTHTHTSHTSHTSHTHTSHTHTSHTSHTHTHTHTHTSHTSHTSH
THTHTSHTHTSHTSHTSHTHTHTSHTSHTSHTHTHTHTHTHTHTHTSHTHTSHTSHTHTHTHTHTSHTSHTSHTSHTHTS
HTHTHTHTHTHTHTSHTSHTHTSHTHTHTSHTSHTSHTHTHTHTSHTHTHTHTHTHTSHTHTHTHTHTSHTHTSHTSH
TSHTSHTHTSHTHTSHTSHTSHTSHTHTSHTHTSHTHTSHTHTHTHTSHTSHTSHTSHTHTSHTHTSHTSHTSHTSHT
HTHTHTHTHTSHTSHTSHTHTHS
```

...we'll talk more about this in a few weeks when we discuss *quantum compilation*.

Measurement: observables and expectation values

Sampling

So far, we've learned how take measurement samples in the computational basis.

```
dev = qml.device('default.qubit', wires=1, shots=100)

@qml.qnode(dev)
def rotate_with_rz(theta):
    qml.Hadamard(wires=0)
    qml.RZ(theta, wires=0)
    return qml.sample()
```

What else can we do?

Measurement outcome probabilities

Compute the measurement outcome probabilities from the results:

```
dev = qml.device('default.qubit', wires=1, shots=100)

@qml.qnode(dev)
def rotate_with_rz(theta):
    qml.Hadamard(wires=0)
    qml.RZ(theta, wires=0)
    return qml.probs()
```

Extract the state

Since we are running on a simulator...

```
# Note that we did NOT specify shots: analytic mode
dev = qml.device('default.qubit', wires=1)

@qml.qnode(dev)
def rotate_with_rz(theta):
    qml.Hadamard(wires=0)
    qml.RZ(theta, wires=0)
    return qml.state()
```

(Can analytically compute probabilities too. But of course we cannot do this with a real device!)

Observables

Generally, we are interested in measuring real, physical quantities. In physics, these are called observables. They are represented by Hermitian matrices. An operator (matrix) H is Hermitian if

$$H = H^\dagger$$

Why Hermitian? The possible measurement outcomes are given by the eigenvalues of the operator, and eigenvalues of Hermitian operators are **real**.

Example:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Z is Hermitian:

Its eigensystem is

Example:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

X is Hermitian and its (normalized) eigensystem is

Example:

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

Y is Hermitian and its (normalized) eigensystem is

When we measure X, Y, or Z on a state, for each shot we will get one of the eigenstates (/eigenvalues). If we take multiple shots, what do we expect to see *on average*?

Analytically, the **expectation value** of measuring the observable M given the state $|\psi\rangle$ is

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

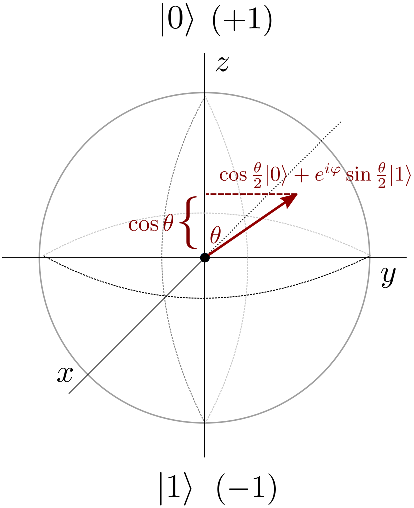
Expectation values: analytical

Example: consider the quantum state

$$|\psi\rangle = \frac{1}{2}|0\rangle - i\frac{\sqrt{3}}{2}|1\rangle.$$

Let's compute the expectation value of Y:

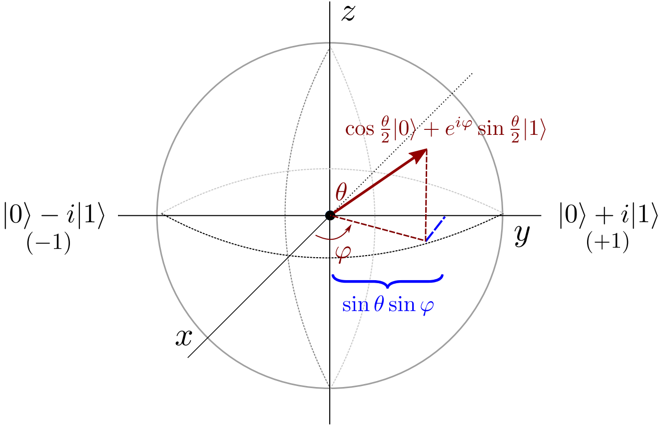
The Bloch sphere offers us some more insight into what a projective measurement is.



$$
\begin{aligned}
|\psi\rangle &= \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle \\
Z|\psi\rangle &= \cos\frac{\theta}{2}|0\rangle - e^{i\varphi}\sin\frac{\theta}{2}|1\rangle \\
\langle\psi|\,Z|\psi\rangle &= \cos^2\frac{\theta}{2} - \sin^2\frac{\theta}{2} \\
&= \cos\theta
\end{aligned}
$$

Expectation values and the Bloch sphere

In this picture, we can visualize measurement in different bases by projecting onto different axes.



Exercise: derive this by computing $\langle\psi| Y|\psi\rangle$.

Expectation values: from measurement data

Let's compute the expectation value of Z for the following circuit using 10 samples:

```
dev = qml.device('default.qubit', wires=1, shots=10)

@qml.qnode(dev)
def circuit():
    qml.RX(2*np.pi/3, wires=0)
    return qml.sample()
```

Results might look something like this:

$$[1, 1, 1, 0, 1, 1, 1, 0, 1, 1]$$

Expectation values: from measurement data

The expectation value pertains to the measured eigenvalue; recall Z eigenstates are

$$\lambda_1 = +1, \qquad |\psi_1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\lambda_2 = -1, \qquad |\psi_2\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

So when we observe $|0\rangle$, this is eigenvalue $+1$ (and if $|1\rangle$, -1). Our samples shift from

```
[1, 1, 1, 0, 1, 1, 1, 0, 1, 1]
```

to

```
[-1, -1, -1, 1, -1, -1, -1, 1, -1, -1]
```

Expectation values: from measurement data

The expectation value is the weighted average of this, where the weights are the eigenvalues:

$$\langle Z \rangle = \frac{1 \cdot n_1 + (-1) \cdot n_{-1}}{N}$$

where

- n_1 is the number of $+1$ eigenvalues
- n_{-1} is the number of -1 eigenvalues
- N is the total number of shots

For our example, $\langle Z \rangle = -0.6$.

Let's do this in PennyLane instead:

```
dev = qml.device('default.qubit', wires=1)

@qml.qnode(dev)
def measure_z():
    qml.RX(2*np.pi/3, wires=0)
    return qml.expval(qml.PauliZ(0))
```

So far we've seen 4 ways of extracting information out of a QNode:

1. `qml.state()`
2. `qml.probs(wires=x)`
3. `qml.sample()`
4. `qml.expval(observable)`

The first three all return results of measurements taken with respect to the computational basis; and most hardware only allows for computational basis measurements. How can we measure with respect to *different bases* with that restriction? (and what does that mean?)

What does it mean to measure in a different bases? Projective measurement with respect to a different set of orthonormal states. For example, $\{|+\rangle, |-\rangle\}$ are an orthonormal basis.

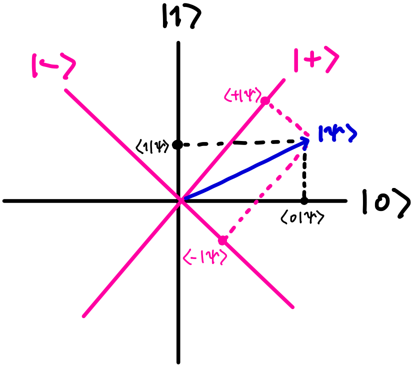


Image credit: Codebook node I.9

Basis rotations

Use a basis rotation to "trick" the quantum computer into measuring in a different basis.

Suppose we want to measure in the Y basis:

$$|i\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + i|1\rangle\right), \quad |-i\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle - i|1\rangle\right).$$

Unitary operations preserve length *and* angles between normalized quantum state vectors.

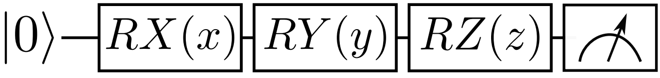
There exists some unitary transformation that will convert between these eigenvectors, and the eigenvectors of Z (the basis in which we will take the measurement).

Let's try to turn

$$|i\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + i|1\rangle\right) \quad \rightarrow \quad |0\rangle$$
$$|-i\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle - i|1\rangle\right) \quad \rightarrow \quad |1\rangle$$

That way, if we measure and observe $|0\rangle$, we know that this was previously $|i\rangle$ in the Y basis (and similarly for $|1\rangle$).

Let's run the following circuit, and measure in the Y basis



- Implement single-qubit quantum algorithms in PennyLane
- Describe the behaviour of common single-qubit gates
- Calculate the expectation value of an observable
- Perform measurements in other bases

What topics did you find unclear today?

Next time

Content:

- Multi-qubit states, operations, and measurements
- Entanglement

Action items:

1. Finish Assignment 0 (due before class Tuesday)
2. Start on Assignment 1 once posted (you can do problem 1)
3. Quiz next class

Recommended reading:

- Codebook nodes I.5-I.10
- Nielsen & Chuang 4.2