

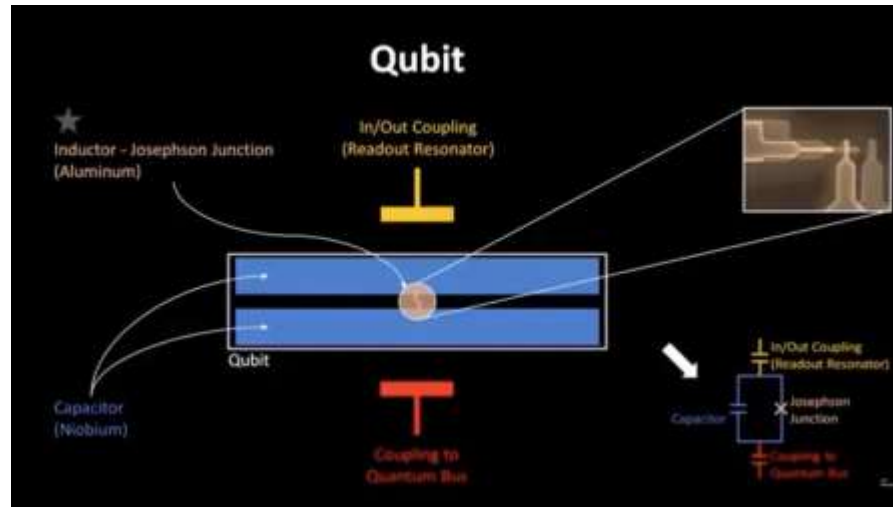
Qiskit Tutorials Workthrough

D. Gayowsky
January 17 2023

Gate-Based Quantum Computing

IBM Qubits: Two superconductors placed on either side of an insulator form a "Josephson Junction", which can hold, charge, and read out individual bits of quantum information. Insulating layer is thin enough to allow weak correlation of superconducting wavefunctions and quantum tunneling through insulating barrier.

"Circuit approach" to designing qubits.



Gate-Based Quantum Computing

IBM is a little vague on exactly what happens in their computers... but, essentially, solves problems in an analogous method to quantum annealing.

Problem solving with gate-based QC:

1. The QC is activated by creating an equal superposition of all possible states.
2. The problem is encoded onto the system by applying **gates**, which affect phase and amplitudes of each state.
3. The QC "comes to a solution by using the physical properties of interference to magnify the amplitude of the correct answer" – (very vague, thank you IBM.)

Where some problems may need repetition of steps 2 and 3.

Quantum Gates

What are quantum gates?

Firstly, imagine a qubit as a complex two-component vector, where α, β are probability amplitudes of a qubit being in states $|0\rangle, |1\rangle$, respectively:

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

Considering the state of a qubit existing on a unit sphere ("Bloch sphere"), we can write:

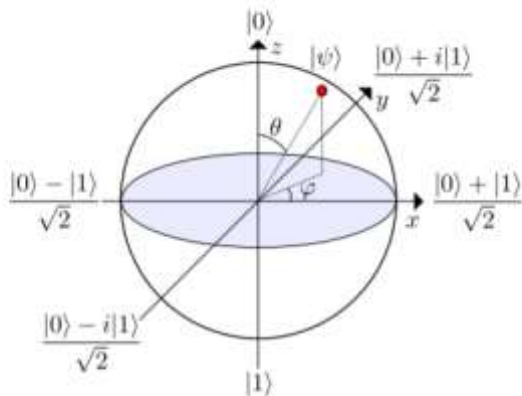


Fig. 1 The Bloch sphere representation of a qubit state. The north pole is the ground state $|0\rangle$ and the south pole is the excited state $|1\rangle$. To convert an arbitrary superposition of $|0\rangle$ and $|1\rangle$ to a point on the sphere, the parametrization $|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle$ is used.

$$\alpha = \cos\left(\frac{\theta}{2}\right)$$
$$\beta = e^{i\phi} \sin\left(\frac{\theta}{2}\right)$$

Quantum gates simply move the point representing a qubit along the surface of the Bloch sphere to a superposition of $|0\rangle, |1\rangle$.

Quantum Gates

For example, we may rotate about the x, y, or z axis using our forever-familiar Pauli matrices, or other quantum gates:










$$X = \sigma_x = \text{NOT} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad |0\rangle \rightarrow |1\rangle, |1\rangle \rightarrow |0\rangle$$

$$Y = \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad |0\rangle \rightarrow i|1\rangle, |1\rangle \rightarrow -i|0\rangle$$

$$Z = \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad |0\rangle \rightarrow |0\rangle, |1\rangle \rightarrow -|1\rangle$$

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad |0\rangle \rightarrow |0\rangle, |1\rangle \rightarrow |1\rangle$$

... and many more, all of which have various effects on the qubit(s) they are operating on.

Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CCNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Hamiltonians

Representing a Hamiltonian: In order to speak in the same "language" as our quantum gates, we wish to represent our Hamiltonian in terms of quantum gates, ie. Pauli x, y, z spin matrices.

In general, transforming a fermionic Hamiltonian to a spin Hamiltonian:

1. Allow the state of each qubit q_i store the occupation of site i (assuming we are working with a chain of sites).
2. Map fermion creation and annihilation operators c_i^+, c_i onto qubit operators.

Example: The Jordan-Wigner transformation uses the following relations:

$$\begin{aligned} c_j^+ &= e^{-i\Phi_j} S_j^+ & S_j^+ &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \frac{1}{2}(\sigma_x + i\sigma_y) & S_j^+ |0\rangle &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |1\rangle & S_j^+ |1\rangle &= 0 \\ c_j &= e^{i\Phi_j} S_j^- & S_j^- &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \frac{1}{2}(\sigma_x - i\sigma_y) & S_j^- |1\rangle &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |0\rangle & S_j^- |0\rangle &= 0 \end{aligned}$$

Gate Based Algorithms

As with classical computing, there are many different gate-based algorithms for different purposes, many of which are simply an improvement to their classical analogue with a smaller order of operations.

Some gate based algorithms include:

- Simon's Algorithm
- Factoring Algorithm
- Grover's Algorithm
- Quantum Fourier Transform
- Quantum Random Walk
- Ad nauseum...

We are interested in the **Hamiltonian simulation** class of algorithms.

Gate Based Algorithms

Hamiltonian Simulation Algorithms:

A class of gate based quantum algorithms which are able to simulate a Hamiltonian of many degrees of freedom.

There are several types of Hamiltonian simulation algorithms:

- **Time dependant** – implementation of time evolution algorithms on a quantum computer.
- **Variational approach** – obtaining approximations to quantum states.
- **Analog quantum systems** – dedicated quantum systems built physically to represent a specific Hamiltonian.

We are not building analog quantum systems ourselves, clearly – unless someone wants to give us a very nice pay raise – so we will disregard those. However, time dependent and variational approaches may be of some use to us.

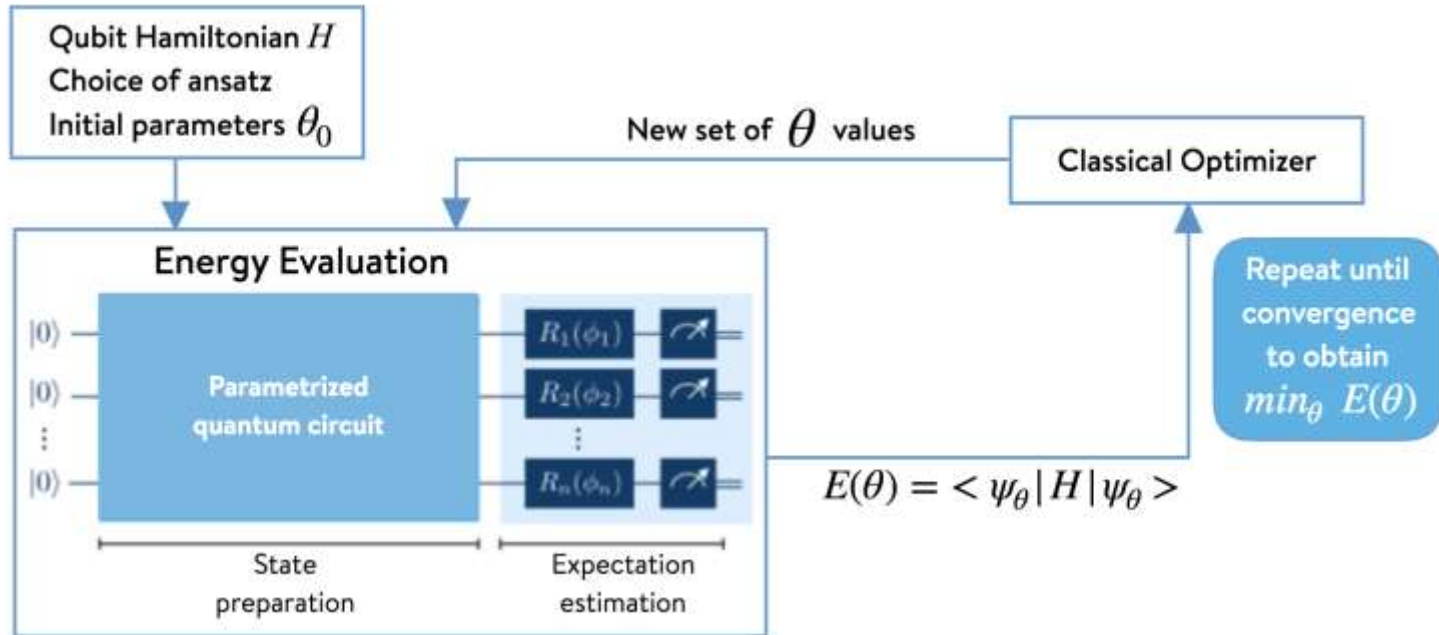
Variational Quantum Eigensolvers

Variational Quantum Eigensolvers (VQEs) are a specific example of the variational approach to quantum computing, widely used for problems like ours – from the name, solving for eigenvalues by implementing both wavefunction preparation ("ansatz") and eigenvalue estimation on a quantum computer.

Generalized VQE Steps (from Qiskit documentation):

1. Make an initial "guess" approximating the ground state, $\psi(\alpha)$.
2. Create a series (circuit) of quantum gates combining to form unitary transformation $U(\theta)$, dependant on some parameters θ .
3. Act with $U(\theta)$ on $\psi(\alpha)$, where the action of this unitary operator, the "ansatz", returns $U(\theta)|\psi(\alpha)\rangle \equiv |\psi(\theta)\rangle$.
4. Classically measure the associated energy of $\psi(\theta)$, $E_\theta \equiv \langle\psi(\theta)|H|\psi(\theta)\rangle$.
5. Classically vary the parameters θ and decide whether to repeat steps 3, 4, minimizing E_θ .

Variational Quantum Eigensolvers



VQE: Input

VQE Input:

In addition to our Hamiltonian, we must choose an easy-to-prepare initial state to feed into our algorithm.

- If the input is encoded in a state "near" to the ground state (ie. $\psi_0 = |0000\rangle, \psi_\alpha = |1000\rangle$), it should not be difficult to collapse into it during measurement.
- If the input is encoded in a state "far" from the ground state, however (ie. $\psi_0 = |0000\rangle, \psi_\alpha = |1011\rangle$), it is equally likely to collapse into any eigenstate of the system... making convergence difficult.

Why we choose to converge to the ground state: The ground state is the "easiest" state to converge to – noise and decoherence can make convergence to excited states unstable.

VQE: Ansatz

Essentially: The ansatz are a series of quantum gates acting on our trial wavefunction, specifically created such that we expect that there is some θ where:

$$U(\theta)|0\rangle = |\psi(\theta)\rangle \approx |\psi_0\rangle$$

The purpose of the ansatz is to prepare the quantum state $|\psi(\theta)\rangle$ for measurement and finding θ . The ansatz constructs an initial state, and builds upon it using the parametrized quantum circuit.

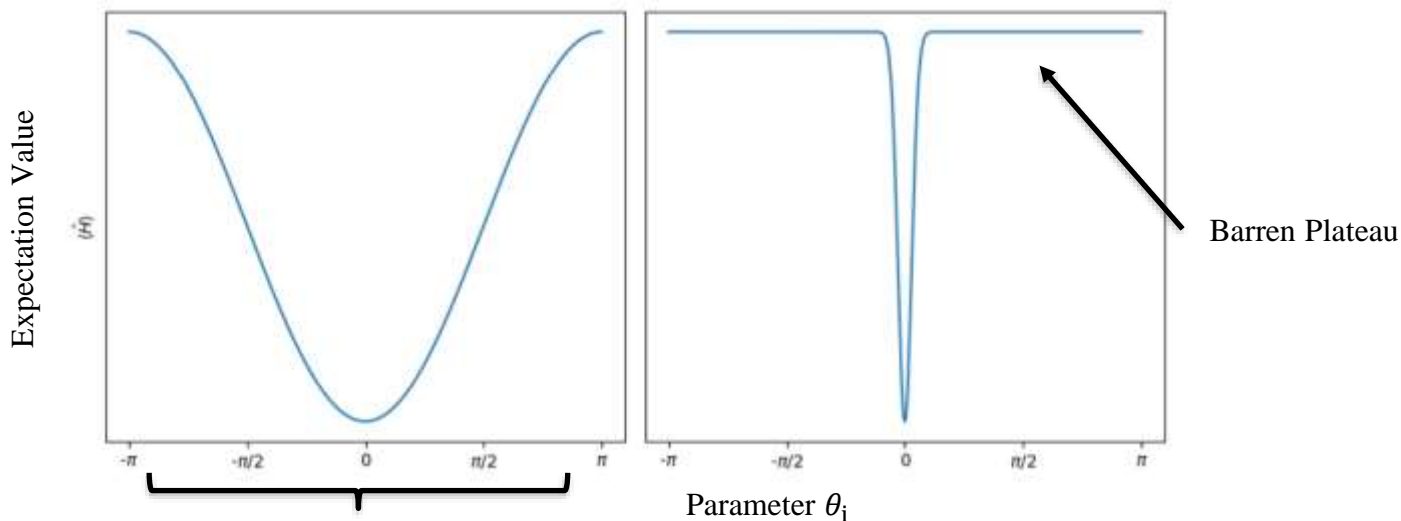
VQEs can be described as "modular" circuits, in which we may substitute or improve specific pieces – this principle also applies to the ansatz. **There is no one single ansatz, and any given ansatz may work for many different Hamiltonians, only a specific class or model of Hamiltonian, etc.**

The choice of ansatz can dramatically improve or destroy VQE convergence – it is the ansatz's job to "guide the optimizer".

VQE: Ansatz

What do we look for in an ansatz?

- Ansatz covers the part of the Hilbert space containing the ground state (or a very near approximation to it). ("Span of possible states" or "Expressibility")
- Ansatz is able to find the optimum set of parameters θ ("The barren plateau problem" or "Trainability")



Span of Possible States about Global Minima

VQE: Ansatz

One Type of Ansatz:

- **Fixed Structure Ansatz** – initialized at beginning of algorithm and remain unchanged throughout the VQE routine, except for variation of parameters. For example, two fixed structure ansatz:

- **Hardware Efficient Ansatz** – sequence of single-qubit rotation gates and entangling gates designed to match hardware without using a large number of gates.

$$|\psi(\theta)\rangle = \left[\prod_{i=1}^d U_{\text{rot}a}(\theta_i) \times U_{\text{ent}} \right] \times U_{\text{rot}a}(\theta_{d+1}) |\psi_{\text{init}}\rangle,$$

- **UCC Ansatz** (Aspuru-Guzik) – evolves an initial wave function under parametrized excitation operators.

$$|\psi\rangle = e^{\hat{T}-\hat{T}^\dagger} |\psi_{HF}\rangle, \quad \hat{T} = \hat{T}_1 + \hat{T}_2 + \dots \hat{T}_\nu, \quad \hat{T}_1 = \sum_{ia} t_i^a \hat{a}_a^\dagger \hat{a}_i, \\ \hat{T}_2 = \sum_{ijab} t_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_j \hat{a}_i.$$

UCC: Example

Example: Aspuru-Guzik Variational Phase Estimation Algorithm (UCC Ansatz)

In each iteration, the Hamiltonian is applied to the qubit register as an exponential time evolution operator:

$$U(\theta) = e^{-iHt}$$

Where the principle is, generally, that if we can evaluate some:

$$e^{-iHt}|\psi(\theta)\rangle = e^{-iE_\theta t}|\psi(\theta)\rangle$$

Then, we measure the "phase factor" $e^{-iE_\theta t}$, and extract E_θ . If we can, with each iteration, converge E_θ closer to E_0 and find the ground state of the system.

VQE: Ansatz

What's the difference?

- HEA is more hardware-efficient, ie. uses a smaller number of qubits and gates.
 - UCC may require some re-allocation of ancilla qubits, management of efficiency, etc.
 - Example: when modeling water molecule, UCC requires ~528 gates and a circuit depth of ~100 steps, whereas HEA requires ~88 gates and a circuit depth of ~40 steps.
- UCC is more accurate, allowing for near-exact parametrization of an arbitrary wavefunction.
 - HEA may not be able to achieve accuracy of results within chemical or floating point precision.

So which kind of ansatz do we use? Well, whichever one is the best model for our Hamiltonian, of course! The ansatz is constructed using bits and pieces of the original Hamiltonian – and an ansatz which does not match our model, no matter how accurate in theory, is not going to converge – thus is no good to anyone.

VQE: Ansatz

Questions:

- Which type of ansatz is better suited to our Kitaev Hamiltonian?
- Is our ansatz of choice scalable for large N sites?
- Do we need to write our own ansatz to treat the Kitaev Hamiltonian? Or is there already one available to us through Qiskit, PennyLane, etc. that will work?
- If we need to design our own ansatz, how can we define and write it for use on a quantum computer?

VQE: Variational Parameters

Variational parameters: The variational parameters θ are, effectively, what we want to change in our circuit in order to determine the ground state or minimal energy of the system.

In Aspuru-Guzik's UCC example, the set of variational parameters θ has a physical meaning – in the Slater orbitals. However, variational parameters can be individual to the problem, and can have many different physical meanings.

For example, in PennyLane we may incorporate the variational parameters θ into a coherent summation of states:

Next, we need to define the quantum circuit that prepares the trial state of the molecule. We want to prepare states of the form,

$$|\Psi(\theta)\rangle = \cos(\theta/2) |1100\rangle - \sin(\theta/2) |0011\rangle,$$

So, we see we have many ways to incorporate variational parameters, and these interact closely with our choice of ansatz.

The Kitaev Hamiltonian for VQE

Recall the fermionic Kitaev Hamiltonian:

$$\hat{H} = t \sum_{i=1}^{N-1} (c_{i+1}^+ c_i + c. c.) + \Delta \sum_{i=1}^{N-1} (c_{i+1}^+ c_i^+ + c. c.) - \mu \sum_{i=1}^N c_i^+ c_i$$

For our Kitaev problem, recall the problem we are trying to solve: we wish to find the ground state configuration and energy of a chain of N sites (to start).

Thus, we can assume that we want to vary the configuration of our possible wavefunctions.

Recall - summing coherently over allowed wavefunctions and possible configurations, written explicitly on three sites:

$$|\varphi\rangle = A_0|0\rangle + A_1|1\rangle + A_2|2\rangle + A_3|3\rangle + A_4|12\rangle + A_5|13\rangle + A_6|23\rangle + A_7|123\rangle$$

The problem we want to solve: What combination or values of A_n will give us the ground state wavefunction?

The Kitaev Hamiltonian for VQE

Thus, to map and solve this problem on a QC we need:

- Write the Kitaev Hamiltonian explicitly in quantum gates (ie. Pauli matrices).
- Find or create a quantum ansatz which successfully solves for the ground state wavefunction and energy.
- Find an optimization routine (and if necessary, define cost function) which successfully solves for the ground state wavefunction and energy.
- Choose a software interface for quantum computing with which to run our simulation.

We already know what our problem is: What combination or values of A_n will give us the ground state wavefunction?

The Kitaev Hamiltonian for VQE

PennyLane Example: Ground state of the hydrogen molecule H2: https://pennyLane.ai/qml/demos/tutorial_vqe.html

```
The Hamiltonian is      (-0.2427450172749822) [Z2]
+ (-0.2427450172749822) [Z3]
+ (-0.04207254303152995) [I0]
+ (0.17771358191549907) [Z0]
+ (0.17771358191549919) [Z1]
+ (0.12293330460167415) [Z0 Z2]
+ (0.12293330460167415) [Z1 Z3]
+ (0.16768338881432715) [Z0 Z3]
+ (0.16768338881432715) [Z1 Z2]
+ (0.17059759240560826) [Z0 Z1]
+ (0.17627661476093917) [Z2 Z3]
+ (-0.04475008421265302) [Y0 Y1 X2 X3]
+ (-0.04475008421265302) [X0 X1 Y2 Y3]
+ (0.04475008421265302) [Y0 X1 X2 Y3]
+ (0.04475008421265302) [X0 Y1 Y2 X3]
```

Here, the Hamiltonian is calculated automatically by PennyLane – however, we can see the formulation of Pauli matrices we need to simulate ours.

We will return to this in later slides...

The Kitaev Hamiltonian for VQE

PennyLane Example: Ground state of the hydrogen molecule H2: https://pennyLane.ai/qml/demos/tutorial_vqe.html

Next, we need to define the quantum circuit that prepares the trial state of the molecule. We want to prepare states of the form,

$$|\Psi(\theta)\rangle = \cos(\theta/2) |1100\rangle - \sin(\theta/2) |0011\rangle,$$

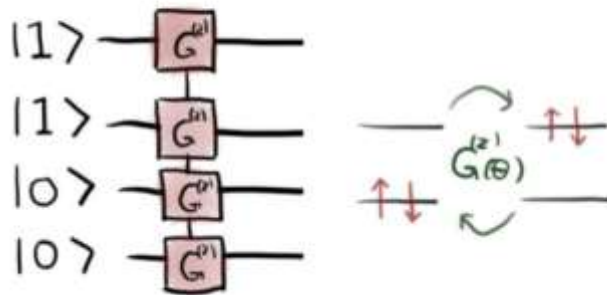
where θ is the variational parameter to be optimized in order to find the best approximation to the true ground state. In the Jordan-Wigner [2] encoding, the first term $|1100\rangle$ represents the Hartree-Fock (HF) state where the two electrons in the molecule occupy the lowest-energy orbitals. The second term $|0011\rangle$ encodes a double excitation of the HF state where the two particles are excited from qubits 0, 1 to 2, 3.

Here, we see they have chosen to **incorporate variational parameters into the wavefunction configuration** – **this is ideal for our model**, as the ground state wavefunction configuration is what we would like to solve for.

The Kitaev Hamiltonian for VQE

PennyLane Example: Ground state of the hydrogen molecule H2: https://pennylane.ai/qml/demos/tutorial_vqe.html

The quantum circuit to prepare the trial state $|\Psi(\theta)\rangle$ is schematically illustrated in the figure below.



Implementing the circuit above using PennyLane is straightforward. First, we use the `hf_state()` function to generate the vector representing the Hartree-Fock state.

```
electrons = 2
hf = qml.qchem.hf_state(electrons, qubits)
print(hf)
```

This is a little more complicated – but I believe what we do here, is map our configurations together...

Are we essentially mapping out our Hamiltonian again if we are considering all possible wavefunction considerations?

The Kitaev Hamiltonian for VQE

PennyLane Example: Ground state of the hydrogen molecule H₂: https://pennyLane.ai/qml/demos/tutorial_vqe.html

The next step is to define the cost function to compute the expectation value of the molecular Hamiltonian in the trial state prepared by the circuit. We do this using the `expval()` function. The decorator syntax allows us to run the cost function as an executable QNode with the gate parameter θ :

```
@qml.qnode(dev)
def cost_fn(param):
    circuit(param, wires=range(qubits))
    return qml.expval(H)
```

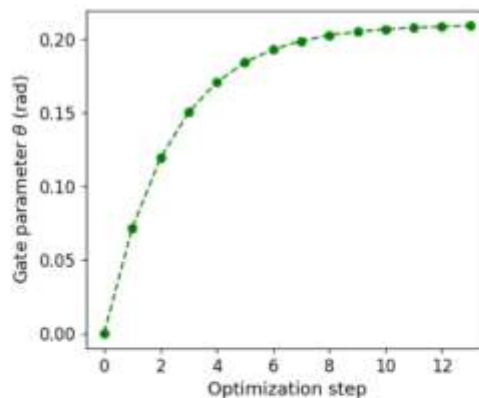
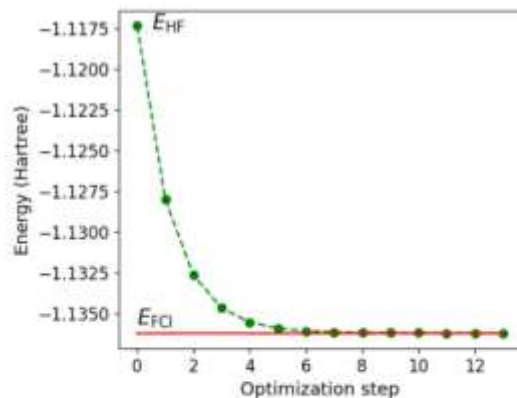
Now we proceed to minimize the cost function to find the ground state of the H₂ molecule. To start, we need to define the classical optimizer. PennyLane offers many different built-in optimizers. Here we use a basic gradient-descent optimizer.

```
opt = qml.GradientDescentOptimizer(stepsize=0.4)
```

Looks here like we are just telling the program that our Hamiltonian is the cost function, and picking an optimizer – nice and easy!

The Kitaev Hamiltonian for VQE

PennyLane Example: Ground state of the hydrogen molecule H₂: https://pennyLane.ai/qml/demos/tutorial_vqe.html



In this case, the VQE algorithm converges after thirteen iterations. The optimal value of the circuit parameter $\theta^* = 0.208$ defines the state

$$|\Psi(\theta^*)\rangle = 0.994 |1100\rangle - 0.104 |0011\rangle,$$

which is precisely the ground state of the H₂ molecule in a minimal basis set approximation.

And then we just
let our code do its
thing – awesome!

The Kitaev Hamiltonian for VQE

Now, following the PennyLane example but with our Kitaev Hamiltonian:

We can, in theory, do the same thing as PennyLane and incorporate variational parameters into our configuration coefficients such that we have some form of:

$$|\varphi(\theta)\rangle = A_0(\theta)|0\rangle + A_1(\theta)|1\rangle + A_2(\theta)|2\rangle + A_3(\theta)|3\rangle + A_4(\theta)|12\rangle + A_5(\theta)|13\rangle + A_6(\theta)|23\rangle + A_7(\theta)|123\rangle$$

Where our cost function will be the expectation value of H , and we will likely use gradient descent optimization as they do here.

Now all we need to do is:

- Write the Kitaev Hamiltonian explicitly in quantum gates (ie. Pauli matrices).
- Find or create a quantum ansatz which successfully solves for the ground state wavefunction and energy.
 - Including: Find general form of $A_n(\theta)$ coefficients – likely sinusoidal(?)

The Kitaev Hamiltonian for VQE

Or:

$$H = \frac{1}{2} X_0 X_1 - \frac{1}{4} Z_0 Y_2 + Z_3.$$

We can directly define this Hamiltonian in PennyLane by summing `qml.Observable` objects and multiplying them by scalars:

```
import pennylane as qml
import pennylane.numpy as np
hamiltonian = 0.5 * qml.PauliX(0) @ qml.PauliX(1) - 0.25 * qml.PauliZ(0) @ qml.PauliY(2) \
+ qml.PauliZ(3)
```

Note that the matrix multiplication operator `@` works as the tensor product \otimes between the Pauli operators. A more general approach for constructing a Hamiltonian in PennyLane is from a list of `qml.Observable` objects and a corresponding list of coefficients:

```
coeffs = [0.5, -0.25, 1.]
obs = [qml.PauliX(0) @ qml.PauliX(1), qml.PauliZ(0) @ qml.PauliY(2), qml.PauliZ(3)]
hamiltonian = qml.Hamiltonian(coeffs, obs)
```

It seems pretty simple to just define our electronic Hamiltonian if we can get it in terms of Pauli X, Y, Z matrices. So, all we have to do to prepare the Kitaev Hamiltonian is make sure we have the correct mapping to our Pauli matrix basis.

The Kitaev Hamiltonian for VQE

We already have our Kitaev Hamiltonian in terms of spin up and spin down operators VIA the Jordan-Wigner transformation. However, we want this in terms of Pauli matrices for use in VQE. Recall:

$$\begin{aligned} S_j^+ &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \frac{1}{2}(\sigma_x + i\sigma_y) & \hat{H} &= t \sum_{j=1}^N (S_{j+1}^+ S_j^- + S_j^+ S_{j+1}^-) - \Delta \sum_{i=1}^N (S_{i+1}^+ S_i^+ + S_i^- S_{i+1}^-) - \mu \sum_{i=1}^N S_i^+ S_i^- \\ S_j^- &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \frac{1}{2}(\sigma_x - i\sigma_y) & \sigma_x \sigma_y &= i\sigma_z, \sigma_z \sigma_x = i\sigma_y, \sigma_y \sigma_z = i\sigma_x \\ \sigma_\alpha^2 &= 1 & \sigma_\alpha \sigma_\beta &= -\sigma_\beta \sigma_\alpha \end{aligned}$$

Let's try using these identities to write our Hamiltonian in terms of x, y, z Pauli matrices as per PennyLane:

This implies that we need to map the fermionic operators onto operators that act on qubits. This can be done by using the [Jordan-Wigner transformation \[5\]](#) which allows us to decompose the fermionic Hamiltonian into a linear combination of the tensor product of Pauli operators

$$H = \sum_j C_j \otimes_i \sigma_i^{(j)},$$

where C_j is a scalar coefficient and σ_i represents an element of the Pauli group $\{I, X, Y, Z\}$.

The Kitaev Hamiltonian for VQE

Where:

This fermionic-to-qubit transformation is done using the `decompose()` function, which uses `OpenFermion` to compute the electron integrals using the previously generated results of the mean field calculation. Then, it builds the fermionic Hamiltonian and maps it to the qubit representation.

```
qubit_hamiltonian = qchem.decompose(hf_file, mapping="jordan_wigner")
print("Qubit Hamiltonian of the water molecule")
print(qubit_hamiltonian)
```

Out:

```
Qubit Hamiltonian of the water molecule
(-46.46390678868894+0j) [] +
(-0.01458364890761264+0j) [X0 X1 Y2 Y3] +
(-3.57076132922807e-07+0j) [X0 X1 Y2 Z3 Z4 Y5] +
```

This specific output Hamiltonian for the water molecule is very long. But if we can use `OpenFermion` to map our Kitaev Hamiltonian automatically to Pauli X, Y, Z matrices, it will save us some time – and ensure our Hamiltonian is in the correct form.

Additionally, if you have built your electronic Hamiltonian independently using `OpenFermion` tools, it can be readily converted to a PennyLane observable using the `convert_observable()` function.

Qiskit Tutorial 1: Circuit Basics

Here, we create a quantum circuit which has three qubits.

We wish to create the **GHZ (Greenberger–Horne–Zeilinger) state**, which is an entangled superposition of quantum states, and is written as:

$$|\psi\rangle = (|000\rangle + |111\rangle)/\sqrt{2}.$$

Can see here that each qubit must be in a superposition of states $|0\rangle$ and $|1\rangle$, like we saw in our Bloch sphere diagram.

To do this via quantum circuit:

1. Start in the empty quantum state, $|0\rangle$.
2. Apply quantum gates as necessary to the three qubits.

$$|\psi\rangle = |a_1 a_2 a_3\rangle \quad |000\rangle = |0\rangle |0\rangle |0\rangle$$

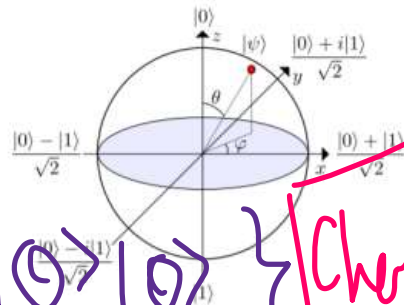


Fig. 1 The Bloch sphere representation of a qubit state. The north pole is the ground state $|0\rangle$ and the south pole is the excited state $|1\rangle$. To convert an arbitrary superposition of $|0\rangle$ and $|1\rangle$ to a point on the sphere, the parametrization $|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle$ is used.

Quantum Gates

For example, we may rotate about the x, y, or z axis using our forever-familiar Pauli matrices, or other quantum gates:









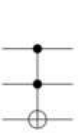
$$X = \sigma_x = \text{NOT} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad |0\rangle \rightarrow |1\rangle, |1\rangle \rightarrow |0\rangle$$

$$Y = \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad |0\rangle \rightarrow i|1\rangle, |1\rangle \rightarrow -i|0\rangle$$

$$Z = \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad |0\rangle \rightarrow |0\rangle, |1\rangle \rightarrow -|1\rangle$$

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad |0\rangle \rightarrow |0\rangle, |1\rangle \rightarrow |1\rangle$$

... and many more, all of which have various effects on the qubit(s) they are operating on.

Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CCNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Qiskit Tutorial 1: Circuit Basics

GHZ (Greenberger–Horne–Zeilinger) state:

$$|\psi\rangle = (|000\rangle + |111\rangle)/\sqrt{2}.$$

$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow$ We can use these to represent multi-qubit states via Kronecker product, defined:

$$\begin{bmatrix} x \\ y \end{bmatrix} \otimes \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} xa \\ xb \\ ya \\ yb \end{bmatrix}, \text{ and we can extend this to any number of qubits... state vector will be of dimension } (1, 2^n).$$

For example, $|101\rangle$ state is then written (transposed, to save space on the slides):

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0]^T$$


$$|cba\rangle = \begin{bmatrix} c_0 b_0 a_0 \\ c_0 b_0 a_1 \\ c_0 b_1 a_0 \\ c_0 b_1 a_1 \\ c_1 b_0 a_0 \\ c_1 b_0 a_1 \\ c_1 b_1 a_0 \\ c_1 b_1 a_1 \end{bmatrix}$$

eg.

Qiskit Tutorial 1: Circuit Basics

How do we act with a single gate on one qubit in a multi-qubit system?

Take the Kronecker product with the identity matrix to expand to the size we want:

$$X \otimes I = \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$


Handwritten note: $I \otimes X \rightarrow \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Example, state $|01\rangle$ applying pauli-X = “not” gate above:

$$\underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{q_1} \otimes \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{q_0} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \rightarrow \underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}}_{\text{on } q_1} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Handwritten labels: q_1 under the first vector, q_0 under the second vector, and $\text{on } q_1$ under the matrix.

Qiskit Tutorial 1: Circuit Basics

GHZ (Greenberger–Horne–Zeilinger) state:

$$|\psi\rangle = (|000\rangle + |111\rangle)/\sqrt{2}.$$

But how do we write an *entangled* state? First look at how the Hadamard gate acts on a single spin down or spin up qubit:

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

$$\hat{H}|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)$$

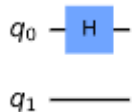
$$\hat{H}|1\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle)$$

Matrix wise, this little matrix is neither our spin up or spin down state – it's a superposition of the two.

Qiskit Tutorial 1: Circuit Basics

Now, how do we expand a single qubit superposition to an entangled state?

Two qubit system: use the Hadamard gate on a single qubit, in our two-qubit system:



Use the Hadamard gate Kronecker product on the state $|01\rangle$:



$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$$= |0\rangle - |1\rangle$$

Qubit 0 always in $|0\rangle$ state, but qubit 1 is in superposition of $|0\rangle$ and $|1\rangle$ states – and we can do this with matrix mechanics.

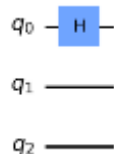
Quantum computer – how do we actually make these gates?

Qiskit Tutorial 1: Circuit Basics

To make the GHZ state, we apply the following gates:

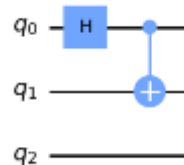
1. A Hadamard gate on qubit 0, which puts it into the superposition state:

```
# Add a H gate on qubit 0, putting this qubit in superposition.
circ.h(0)
circ.draw(output='mpl')
plt.show()
```



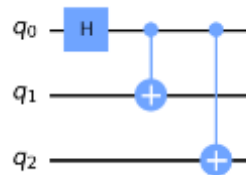
2. A Controlled-NOT operation () between qubit 0 and qubit 1.

```
# Add a CX (CNOT) gate on control qubit 0 and target qubit 1, putting
# the qubits in a Bell state.
circ.cx(0, 1)
circ.draw(output='mpl')
plt.show()
```



3. A Controlled-NOT operation between qubit 0 and qubit 2.

```
# Add a CX (CNOT) gate on control qubit 0 and target qubit 2, putting
# the qubits in a GHZ state.
circ.cx(0, 2)
circ.draw(output='mpl')
plt.show()
```

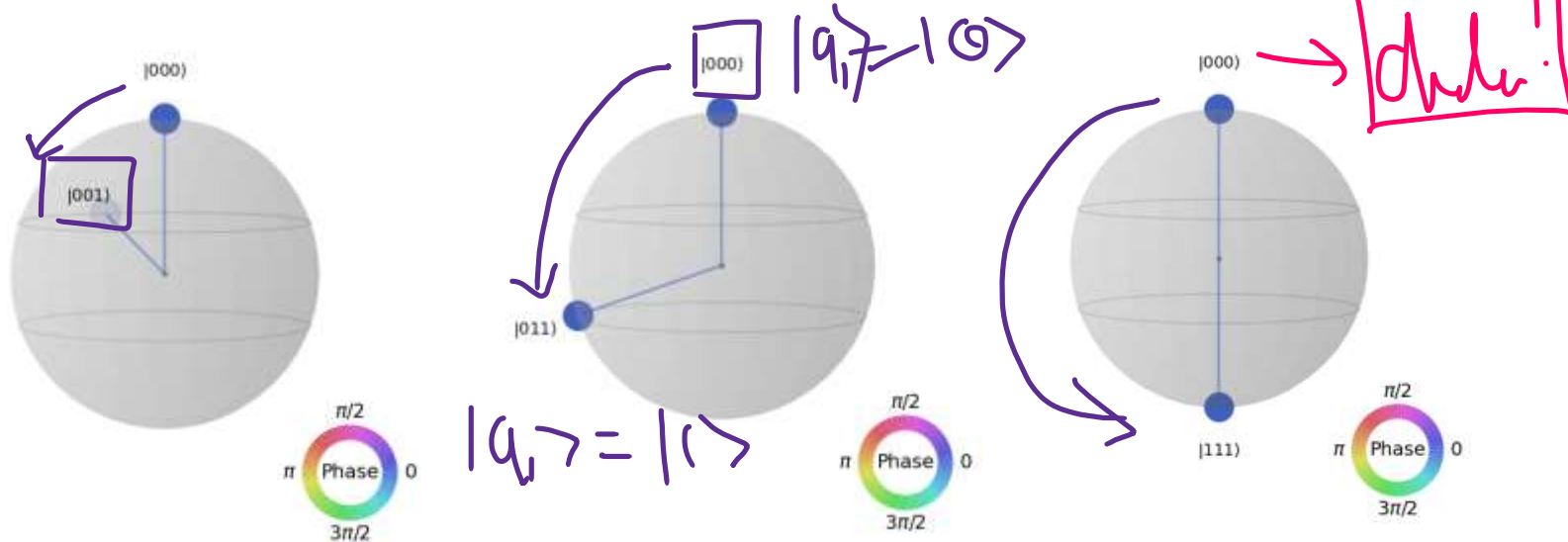


Qiskit Tutorial 1: Circuit Basics

To make the GHZ state, we apply the following gates:

1. A Hadamard gate on qubit 0, which puts it into the superposition state,
2. A Controlled-NOT operation () between qubit 0 and qubit 1.
3. A Controlled-NOT operation between qubit 0 and qubit 2.

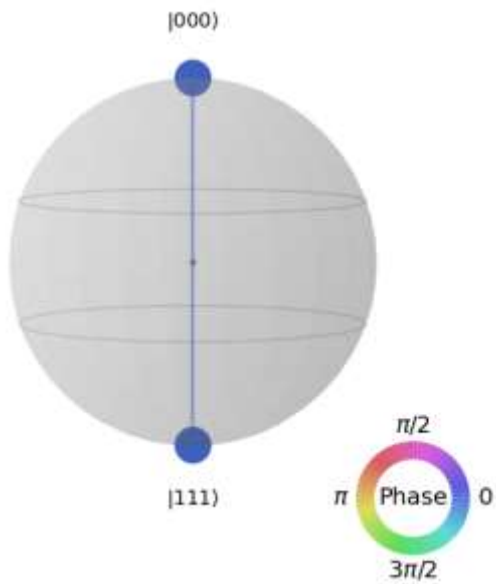
$$|q_2 q_1 q_0\rangle$$



Qiskit Tutorial 1: Circuit Basics

What about taking measurements? At any given time, we might like to know what state our system is in.

When we run this circuit, and take a measurement, recall quantum superposition – **we have to collapse into some single state, we won't actually *observe* a superposition.**



Purely by observation, at the end of our circuit our system is in the superposition of states $|000\rangle$ and $|111\rangle$.

So, when we actually measure or “observe” the system, we will fall into one of these states.

Specifically, for GHZ, the probability to observe either of these states is $\frac{1}{2}$.

Qiskit Tutorial 1: Circuit Basics

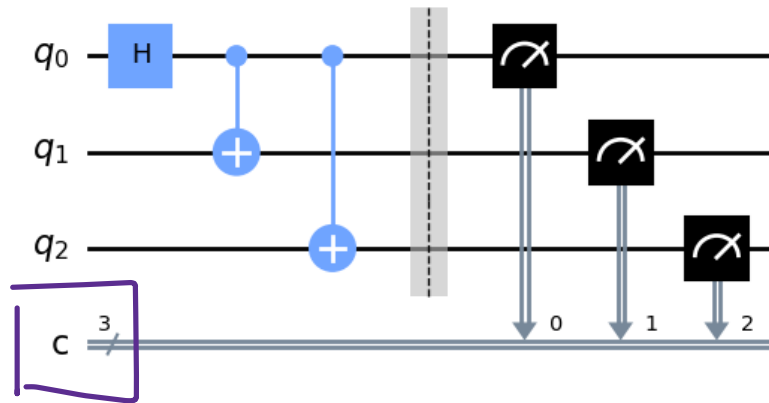
To take the measurement(s):

1. Add a classical register to the circuit.
2. Take three measurements, one of each qubit (which will either be $|0\rangle$ or $|1\rangle$).
3. Output measurements to bits.

```
meas = QuantumCircuit(3, 3)
meas.barrier(range(3))
# map the quantum measurement to the classical bits
meas.measure(range(3), range(3))

# The Qiskit circuit object supports composition.
# Here the meas has to be first and front=True (putting it before)
# as compose must put a smaller circuit into a larger one.
qc = meas.compose(circ, range(3), front=True)

#drawing the circuit
qc.draw('mpl')
plt.show()
```



Qiskit Tutorial 1: Circuit Basics

Measurement Results:

Now if we actually run our circuit and read out our results, what do we get?

```
backend = AerSimulator()

# First we have to transpile the quantum circuit
# to the low-level QASM instructions used by the
# backend
qc_compiled = transpile(qc, backend)

# Execute the circuit on the qasm simulator.
# Default number of 'shots' is 1024.
job_sim = backend.run(qc_compiled, shots=1)

# Grab the results from the job.
result_sim = job_sim.result()
counts = result_sim.get_counts(qc_compiled)
print(counts)
```

Number of runs with measurement.

Joint measurement? $\rightarrow \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \end{matrix}$

```
{ '111': 1 }
```

For this particular run, we measure the state $|111\rangle$.

If we set our shots to default (1024), we should see each of $|111\rangle$ and $|000\rangle$ approximately half the time...

```
{ '000': 520, '111': 504 }
```

Which we do! (With some standard deviations.)

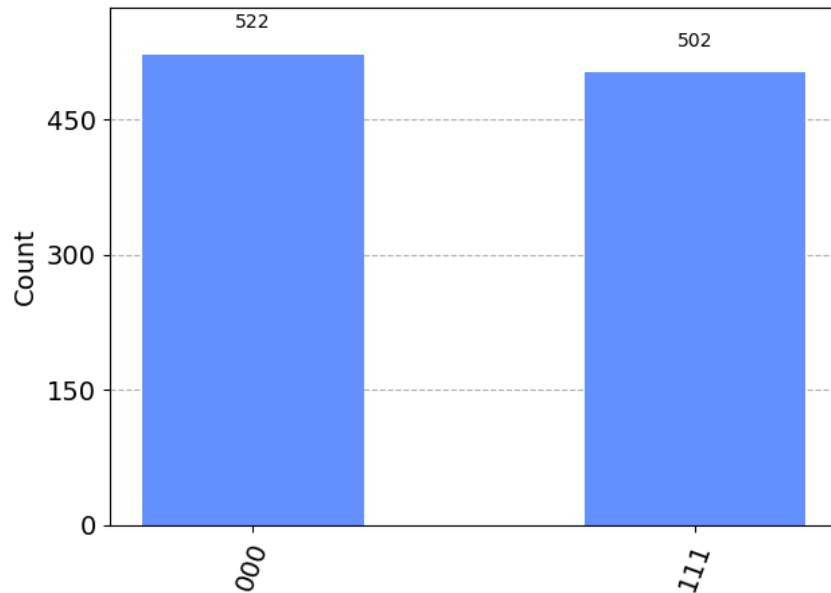
Real \rightarrow distributed, noise!

$|111\rangle, |000\rangle$
 $|q_1\rangle = |11\rangle$

Qiskit Tutorial 1: Circuit Basics

Measurement Results:

And we can show this kind of count result in a histogram:



Qiskit Tutorial 1: Circuit Basics

Overall Observations:

- Can very easily build a quantum circuit to put our system into a superposition of states.
- Can then measure superposition.
- Distribution of measurements will model probability distribution function for each superimposed state.
- **Note: this is a simulation of an ideal quantum circuit – real quantum circuits will have other issues such as phase error, noise, etc. which will affect measurement results.**

Further questions:

- How large of a system can we do this with? Bounded by physical restrictions? Noise?
- Can we use ML methods to predict the series of gates we need to model a certain system?
- Does accuracy decrease with system size? Or is it based on the complexity of the system? Not at all?

Qiskit Tutorial 2: Noise

What happens when we include quantum noise?

Consider again our GBZ state of three qubits above. We can have three kinds of errors, from Qiskit:

- **Single-qubit gate errors** consisting of a single qubit depolarizing error followed by a single qubit thermal relaxation error.
- **Two-qubit gate errors** consisting of a two-qubit depolarizing error followed by single-qubit thermal relaxation errors on both qubits in the gate.
- **Single-qubit readout errors** on the classical bit value obtained from measurements on individual qubits.

Depolarizing error: random Pauli gate is applied to a qubit or set of qubits, inducing phase or state error. Can disentangle qubits.

Thermal relaxation error: phase amplitude damping error, **affects phases of qubits.**

Readout error: Incorrectly reading qubit state by classical computer, caused by classical noise and temp. fluc.

Qiskit Tutorial 2: Noise

Qiskit Aer device noise model automatically generates a simplified noise model. Takes into account:

- Gate errors,
- Thermal relaxation errors,
- Readout errors.

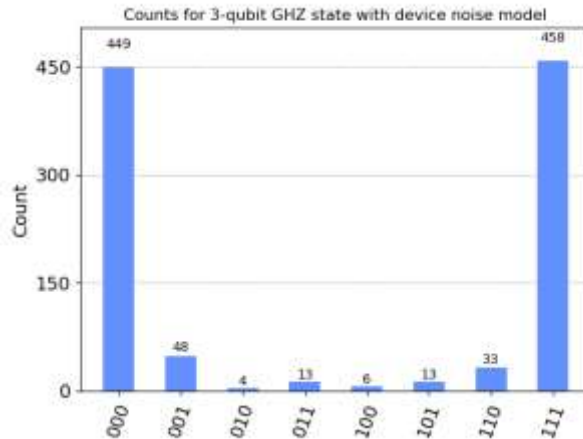
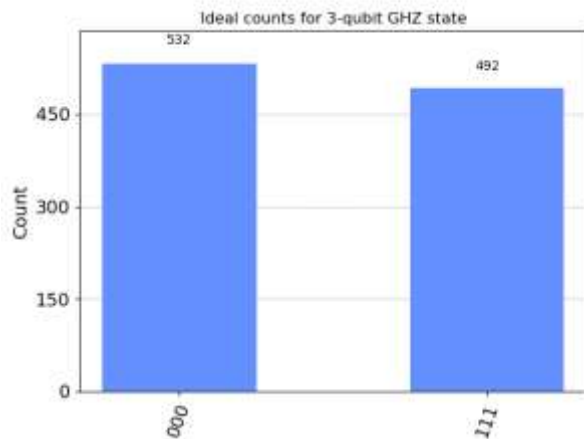
These are probabilistic in nature, and are generated based on real noise data for some IBM device, stored in Qiskit.

Here, this tutorial uses IBM Vigo.*

*Noise data and resultant noise in our simulation is generated by various Aer functions – would need to look into for more detailed analysis.

Qiskit Tutorial 2: Noise

Now, repeating our GHZ simulation: Shown with and without noise. Essentially, we repeat the tutorial 1 – but this time, we create the simulated “device”, and store its noise properties before sending the circuit to device.



Noise resulting in loss of entanglement, bit flip error, etc. results in additional states being measured, rather than just $|000\rangle$ and $|111\rangle$ states as expected.

Can Qiskit tell us which source of noise accounts for which resultant states? Sure it's tracked somewhere, not sure how to get to the data... feed in features, run simulation, collect samples on certain noise type.

Vector of 8 numbers to decision tree → try and predict what noise was active for that experiment.

Qiskit Tutorial 2: Noise

Further work on noise:

- Generating custom noise (building noise models – can define what types of noise we want, how much, etc.)
- Figure out generalized noise stats for quantum machines (what's normal? 5%? 10%? What is our pdf of noise and what types?)
- Look more at sources of noise – usually physical, need to figure out exactly what is occurring, maths behind it.
- Approximating noise with other types of noise – not something I thought existed, but apparently...

Qiskit Tutorial 3: Types & Custom Noise

Applying noise to circuits:

Here we have to do a few things to apply noise:

1. Define the noise probability:

```
# Example error probabilities
p_reset = 0
p_meas = 0
p_gate1 = 0.2
```

2. Create the error, e.g. what exactly happens to the qubit:

```
# QuantumError objects
error_reset = pauli_error([('X', p_reset), ('I', 1 - p_reset)])
error_meas = pauli_error([('X', p_meas), ('I', 1 - p_meas)])
error_gate1 = pauli_error([('X', p_gate1), ('I', 1 - p_gate1)])
error_gate2 = error_gate1.tensor(error_gate1)
```

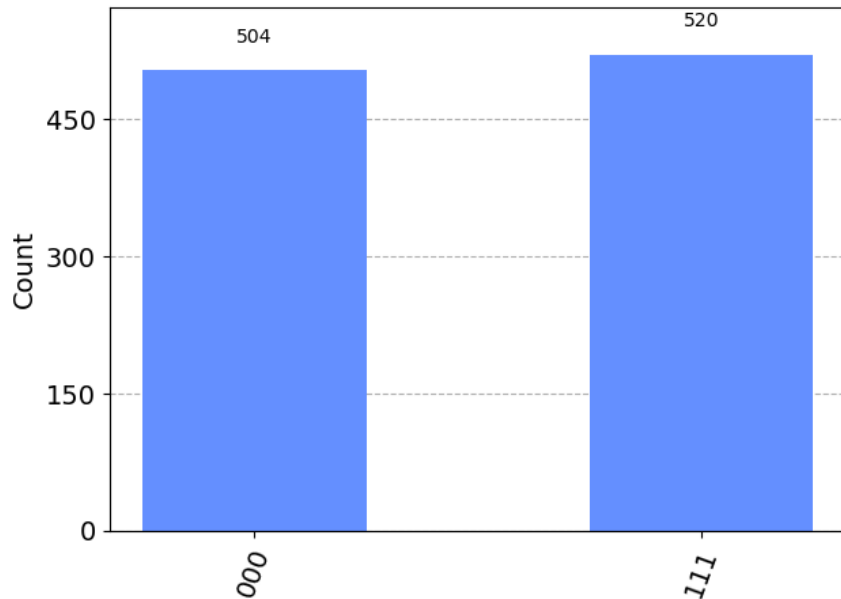
3. Specifically define which gates or process we want the noise to apply to:

```
# Add errors to noise model
noise_bit_flip = NoiseModel()
noise_bit_flip.add_all_qubit_quantum_error(error_reset, "reset")
noise_bit_flip.add_all_qubit_quantum_error(error_meas, "measure")
noise_bit_flip.add_all_qubit_quantum_error(error_gate1, ["h"])
noise_bit_flip.add_all_qubit_quantum_error(error_gate2, ["cx"])
```

Qiskit Tutorial 3: Types & Custom Noise

Now, repeating our GHZ simulation: Show different types of noise.

Reset error: When resetting a qubit (to ground state) reset to 1 instead of 0 with probability p_{reset} .

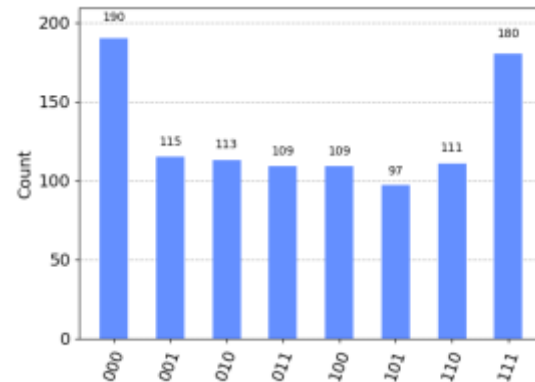
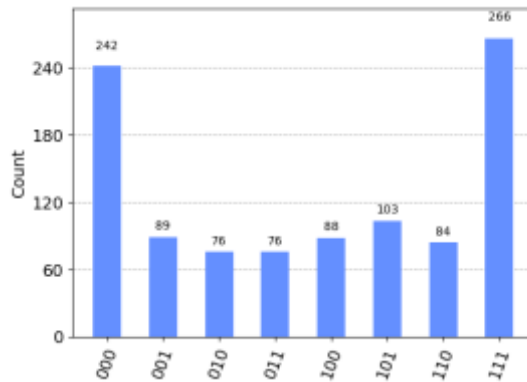
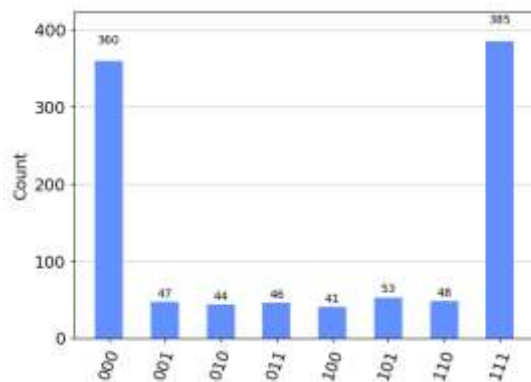


Here reset error doesn't have any effect – we don't actually reset qubits at any point in our circuit!

Qiskit Tutorial 3: Types & Custom Noise

Now, repeating our GHZ simulation: Show different types of noise.

Measurement error: When measuring a qubit, flip the state of the qubit with probability p_{meas} .



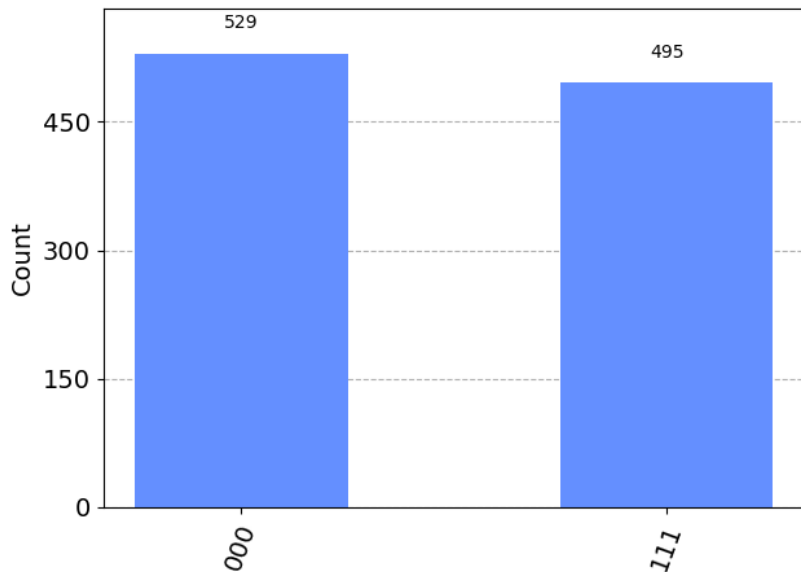
Results when considering GHZ state with $p_{meas} = 0.1, 0.2, 0.3$, respectively.

We can see that as we increase probability of measurement error, we do indeed see a greater percentage of mis-measured qubits.

Qiskit Tutorial 3: Types & Custom Noise

Now, repeating our GHZ simulation: Show different types of noise.

Gate error: When applying a single qubit gate, flip the state of the qubit with probability p_{gate} .



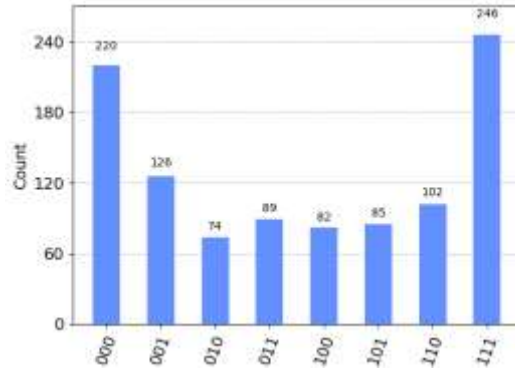
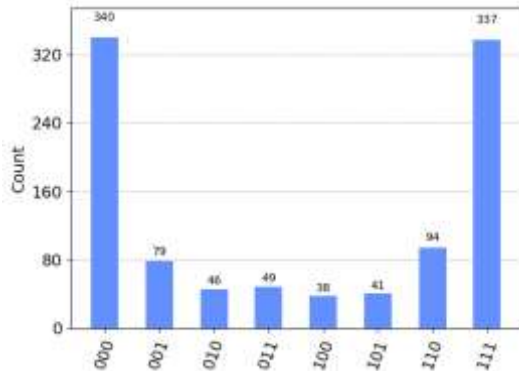
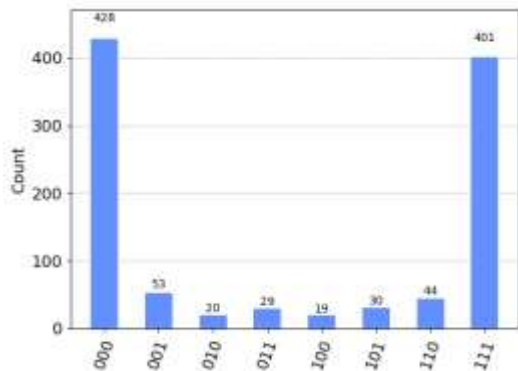
Here single-gate error doesn't really have any effect either, as our only single-qubit operation is the Hadamard gate...

Need to ask Khabat or Stefanie why Hadamard gate appears to be fault-tolerant... perhaps no effect as we're throwing the qubit into a superposition, so single qubit error just changes $|000\rangle$, $|111\rangle \rightarrow |111\rangle$, $|000\rangle$, and our results don't change...?

Qiskit Tutorial 3: Types & Custom Noise

Now, repeating our GHZ simulation: Show different types of noise.

Gate error: When applying a 2-qubit gate, apply single-qubit errors to each qubit.



Results when considering GHZ state with $p_{gate1} = 0.05, 0.1, 0.2$, respectively.

As we increase probability of gate error, we do indeed see a greater percentage of mis-measured qubits.

Qiskit Tutorial 3: Types & Custom Noise

Observations:

- Which type of noise can occur depends on the circuit, e.g. single qubit gate errors cannot occur on circuits with no single qubit gates!
- Some gates may be fault-tolerant, meaning even regular noise does not act on them – however, this is rare...
- Noise often “overlaps”, e.g. multiple types of noise can create the same “incorrect” state.

This leads us to...

Qiskit Tutorial 3: Making a DNN

DNN to Detect Quantum Noise: Thinking about this... may not really work.

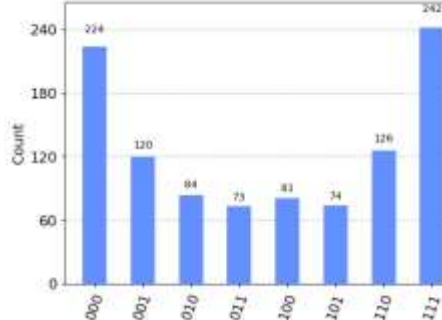
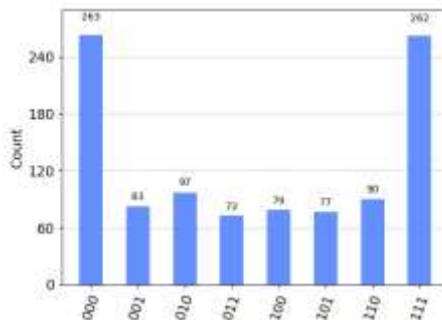
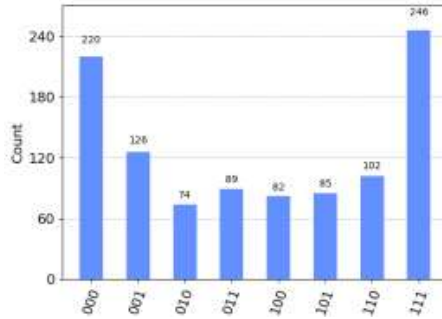
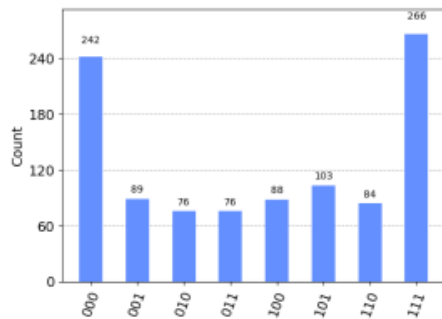
We need to have enough differential in our noise so that we can classify into different types.

However, a couple issues with this...

- Multiple types of noise can result in the same observational readouts...
- Which look very similar in distribution, thus it can be hard or even impossible to distinguish what kind of noise is occurring.
- Noise type is dependent on circuit – i.e. some circuits cannot have some types of noise, thus, any DNN would purely be for a single circuit rather than generalized – would that be useful?

Qiskit Tutorial 3: Making a DNN

DNN to Detect Quantum Noise: Example:



$p_{meas} = 0.2$ vs $p_{gate1} = 0.2$

Question is:

Comparing these, we can see there is a definite pattern with the two-qubit gate error (i.e. when acting with CNOT gate, we can see we are generally more likely to have errors in the 001 and 110 state)...

But is there enough of a difference for a classification algorithm to discern? What about on other circuits? What about larger circuits?

Making a DNN

DNN to Detect Quantum Noise: Data Generation

How do we generate and use the data for our DNN?

1. We run the simulation, and are given a dictionary of states and corresponding counts.
2. Run the simulation multiple times.
3. Compute the mean and standard deviation of each state for all combined trials.
4. Repeat this for $p_{meas} = \{0, 0.1, 0.2, 0.3\}$ and $p_{gate} = \{0, 0.1, 0.2, 0.3\}$.
5. Store data in pandas dataframe.
6. Store data in SQL database.
7. Visualize data by creating plots, e.g. frequency of state at each p_{meas}, p_{gate} .

Data Generation & Management

Dataframe structure: what's the best way to structure this for feeding to a neural network?

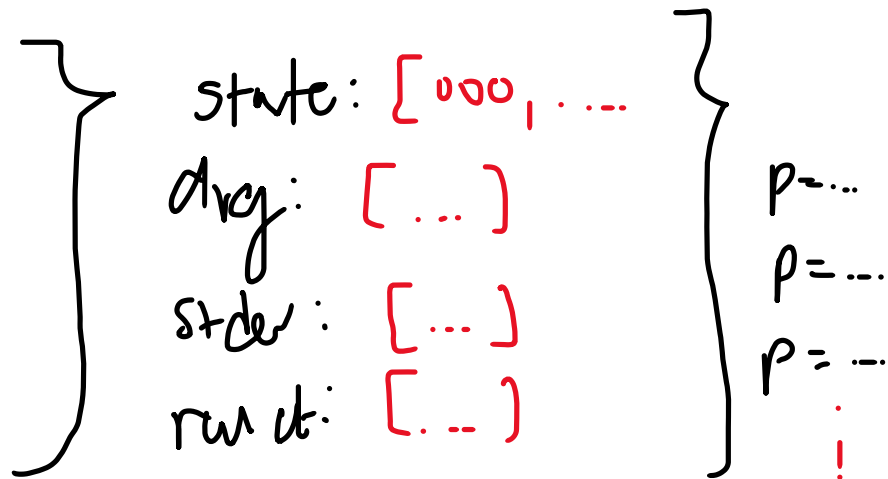
```
import pandas as pd
```

```
data = {  
    "calories": [420, 380, 390],  
    "duration": [50, 40, 45]  
}
```

```
#load data into a DataFrame object:
```

```
df = pd.DataFrame(data)
```

```
print(df)
```



repeat for each 'p'!

State	avg
a	b
c	d
e	f
!	!

Data Generation & Management

State	p_meas	p_gate	Standard Deviation	Mean
000	a	b	c	e
001	a	b	d	f
010
011				
100				
101				
110				
111				

Dataframe structure, in general....

Repeat ad nauseum for all possible combinations of p_meas and p_gate.

Data Generation & Management

```
#Generate error probability values:
meas_prob_vals = np.linspace(min_meas_prob, max_meas_prob, num_vals)
gate_prob_vals = np.linspace(min_gate_prob, max_gate_prob, num_vals)

tot_means = []
tot_stddevs = []
tot_labels = []
tot_meas_prob = []
tot_gate_prob = []

#Generate data for each combination of measurement and gate errors:
for i in range(num_vals):

    #Now we want to pop all this stuff into a data frame...
    noise_data = pd.DataFrame()
    noise_data['State'] = pd.Series(tot_labels)
    noise_data['Meas. Err. Prob'] = pd.Series(tot_meas_prob)
    noise_data['Gate Err. Prob'] = pd.Series(tot_gate_prob)
    noise_data['Count Mean'] = pd.Series(tot_means)
    noise_data['Count Std. Dev.'] = pd.Series(tot_stddevs)
```

	State	Meas. Err. Prob	Gate Err. Prob	Count Mean	Count Std. Dev.
0	000	0.0	0.0	50.1	4.482187
1	001	0.0	0.0	0.0	0.000000
2	010	0.0	0.0	0.0	0.000000
3	011	0.0	0.0	0.0	0.000000
4	100	0.0	0.0	0.0	0.000000
..
67	011	0.2	0.2	10.8	2.712932
68	100	0.2	0.2	10.3	3.257299
69	101	0.2	0.2	10.0	3.000000
70	110	0.2	0.2	12.2	3.218695
71	111	0.2	0.2	16.3	3.318132

Here we create our possible error probability values, then iterate over them.

Generate our noise data and pop into data frame... which we can read out like so.

Note: here we also add in “zero counts” for all possible quantum states which do not appear in our simulation.

Data Generation & Management

```
#Write a function to create an sql database and send data to database:
def noise_data_to_database(min_meas_prob, max_meas_prob, min_gate_prob, max_gate_prob):
    #Use our previous function to create our noise data:
    noise_data = generate_all_error_data(min_meas_prob, max_meas_prob, min_gate_prob)

    #Create in-memory sql database:
    sql_engine = create_engine('sqlite://', echo=False)
    #make sure we're connected to the database...
    connection = sql_engine.raw_connection()
    #send dataframe to sql:
    noise_data.to_sql('quantum_noise_data', connection, if_exists='replace')

    #grab our raw results...
    results = connection.execute("SELECT * FROM quantum_noise_data").fetchall()
    print(results)
```

```
((0, '000', 0.0, 0.0, 50.2, 4.667661580737736), (1, '001', 0.0, 0.0, 0.0, 0.0), (2, '010', 0.0, 0.0, 0.0, 0.0), (3, '011', 0.0, 0.0, 0.0, 0.0), (4, '100', 0.0, 0.0, 0.0, 0.0), (5, '101', 0.0, 0.0, 0.0, 0.0), (6, '110', 0.0, 0.0, 0.0, 0.0), (7, '111', 0.0, 0.0, 80.2, 4.467661580737736), (8, '000', 0.0, 0.1, 33.0, 5.31899356823891), (9, '001', 0.0, 0.1, 7.2, 2.7129318932501877), (10, '010', 0.0, 0.1, 4.1, 2.3817848568466931), (11, '011', 0.0, 0.1, 4.9, 1.9089372712288567), (12, '100', 0.0, 0.1, 3.7, 1.808877584449522), (13, '101', 0.0, 0.1, 4.3, 2.0024984384588757), (14, '110', 0.0, 0.1, 8.5, 2.24722695454531), (15, '111', 0.0, 0.1, 34.3, 3.484381841912286), (16, '000', 0.0, 0.2, 20.5, 3.772917217635375), (17, '001', 0.0, 0.2, 12.3, 2.147091033354389), (18, '010', 0.0, 0.2, 2.2, 2.6756178138651307), (19, '011', 0.0, 0.2, 7.8, 1.9380739424665317), (20, '100', 0.0, 0.2, 7.9, 2.888943816817620), (21, '101', 0.0, 0.2, 8.0, 1.8978662943100751), (22, '110', 0.0, 0.2, 12.2, 3.3486588176088135), (23, '111', 0.0, 0.2, 28.1, 4.321823644348277), (24, '000', 0.1, 0.0, 32.2, 2.673818120931807), (25, '001', 0.1, 0.0, 3.2, 1.8888097862117809), (26, '010', 0.1, 0.0, 5.1, 2.431530134482523), (27, '011', 0.1, 0.0, 5.3, 0.0), (28, '100', 0.1, 0.0, 3.6, 1.962060915181311), (29, '101', 0.1, 0.0, 4.4, 2.0981868261974), (30, '110', 0.1, 0.0, 4.9, 2.07123317220988), (31, '111', 0.1, 0.0, 35.0, 4.548126676494218), (32, '000', 0.2, 0.1, 24.8, 4.5000000000000005), (33, '001', 0.1, 0.1, 18.7, 2.786622844026744), (34, '010', 0.1, 0.1, 8.0, 1.8130302779811362), (35, '011', 0.1, 0.1, 0.1, 2.682705301138809), (36, '100', 0.1, 0.1, 0.1, 2.3823728881753127), (37, '101', 0.1, 0.1, 7.6, 1.9823880098675317), (38, '110', 0.1, 0.1, 8.6, 3.681738187411326), (39, '111', 0.1, 0.1, 27.1, 4.608255719567784), (40, '000', 0.1, 0.2, 19.6, 3.2623801396060318), (41, '001', 0.1, 0.2, 11.4, 2.5615898802071190), (42, '010', 0.1, 0.2, 9.2, 3.1180951878862103), (43, '011', 0.1, 0.2, 9.0, 1.8890802511576748), (44, '100', 0.1, 0.2
```

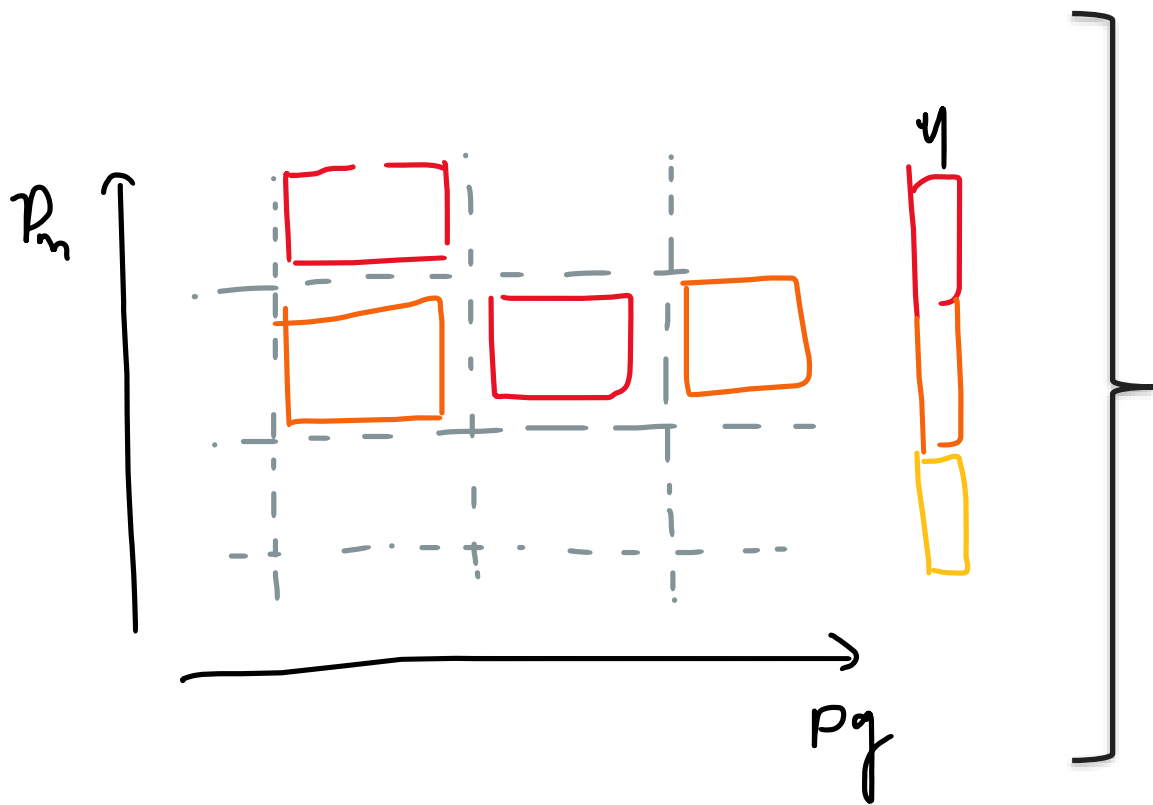
Index	State	Meas. Err. Prob	Gate Err. Prob	Count Mean	Count Std. Dev.
0	000	0.0	0.0	49.0	2.211334
1	001	0.0	0.0	0.0	0.000000
2	010	0.0	0.0	0.0	0.000000
3	011	0.0	0.0	0.0	0.000000
4	100	0.0	0.0	0.0	0.000000
...
57	011	0.2	0.2	12.1	3.112876
68	100	0.2	0.2	12.1	2.981618
69	101	0.2	0.2	11.4	3.352611
70	110	0.2	0.2	11.3	3.716181
71	111	0.2	0.2	15.4	2.973234

Similarly, can generate our results, then send our pandas dataframe to a temporary memory based SQL database....

Which if we read out directly from the database, we just get a bunch of tuples etc., but we can again read that database into a pandas dataframe.

Which looks a lot nicer again!

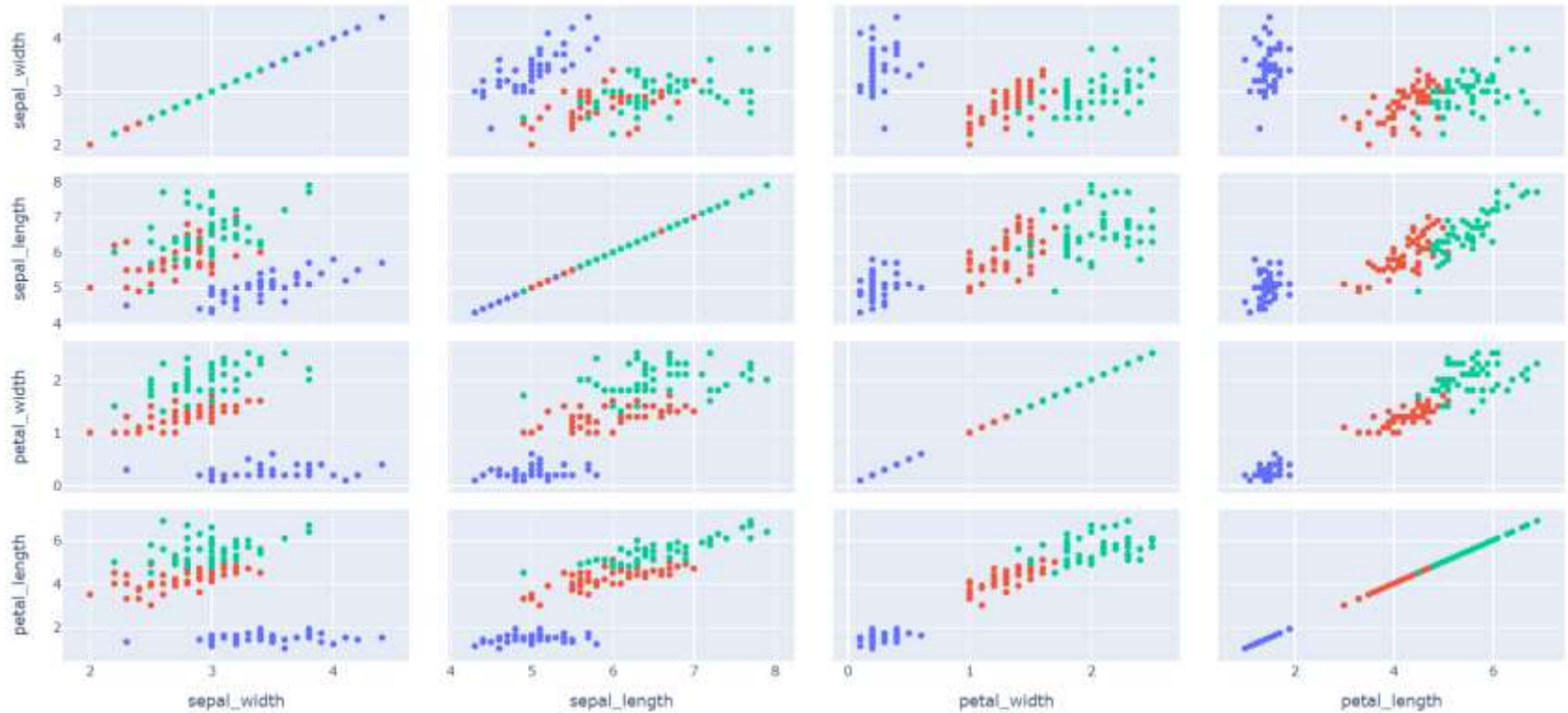
Data Visualization



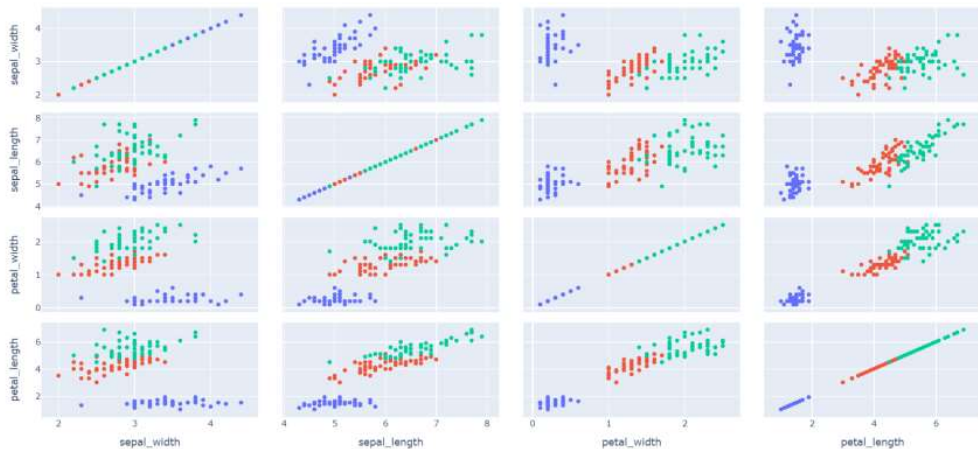
Want to visualize our data...
how does the standard
deviation, frequency, and
mean of each possible state
change as we vary our error
probabilities?

Best way to do this... contour
plot of some kind? Is that even
useful? Potentially better way
to do this?

Data Visualization



Data Visualization



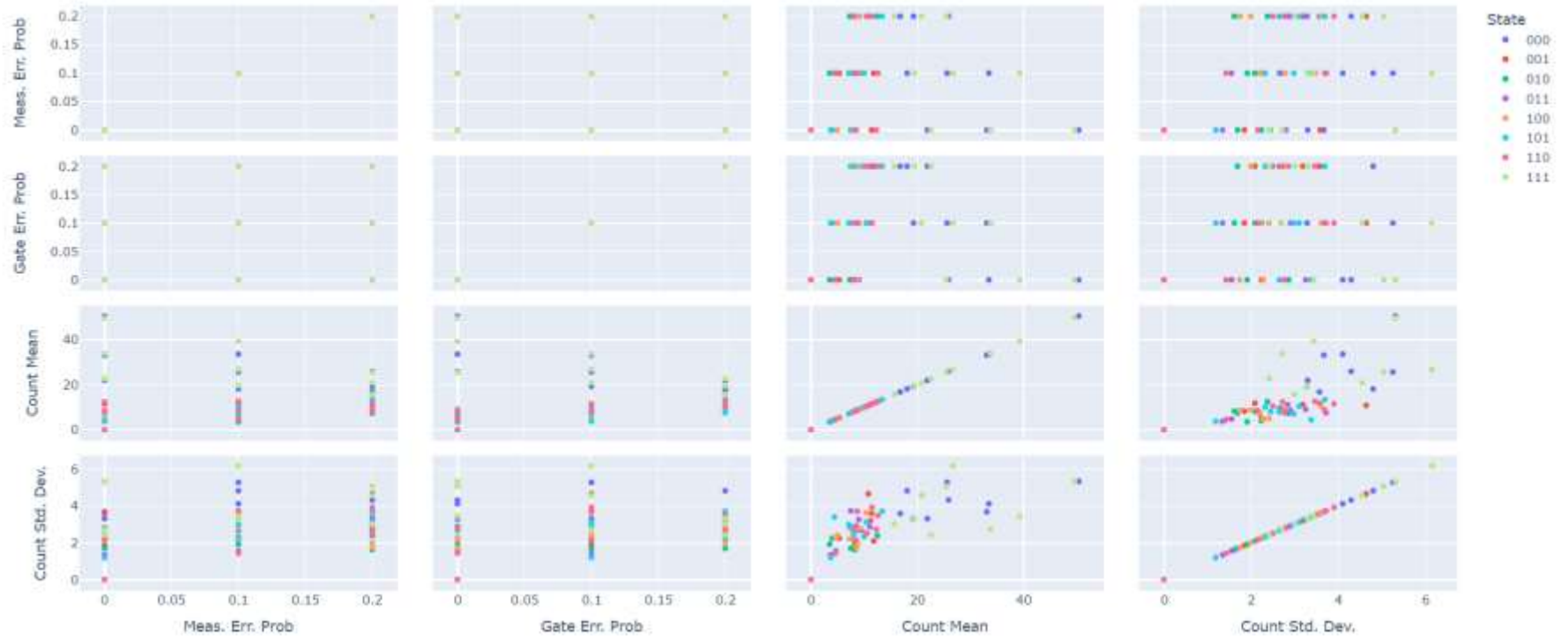
P_meas				
P_gate				
stdev				
mean				
	mean	stdev	P_gate	P_meas

Can do similar thing here...

- 8 possible states (corresponding to 3 kinds of flowers)
- As p_meas increases, look at stdev and mean increase
- As p_gate increases, look at stdev and mean increase

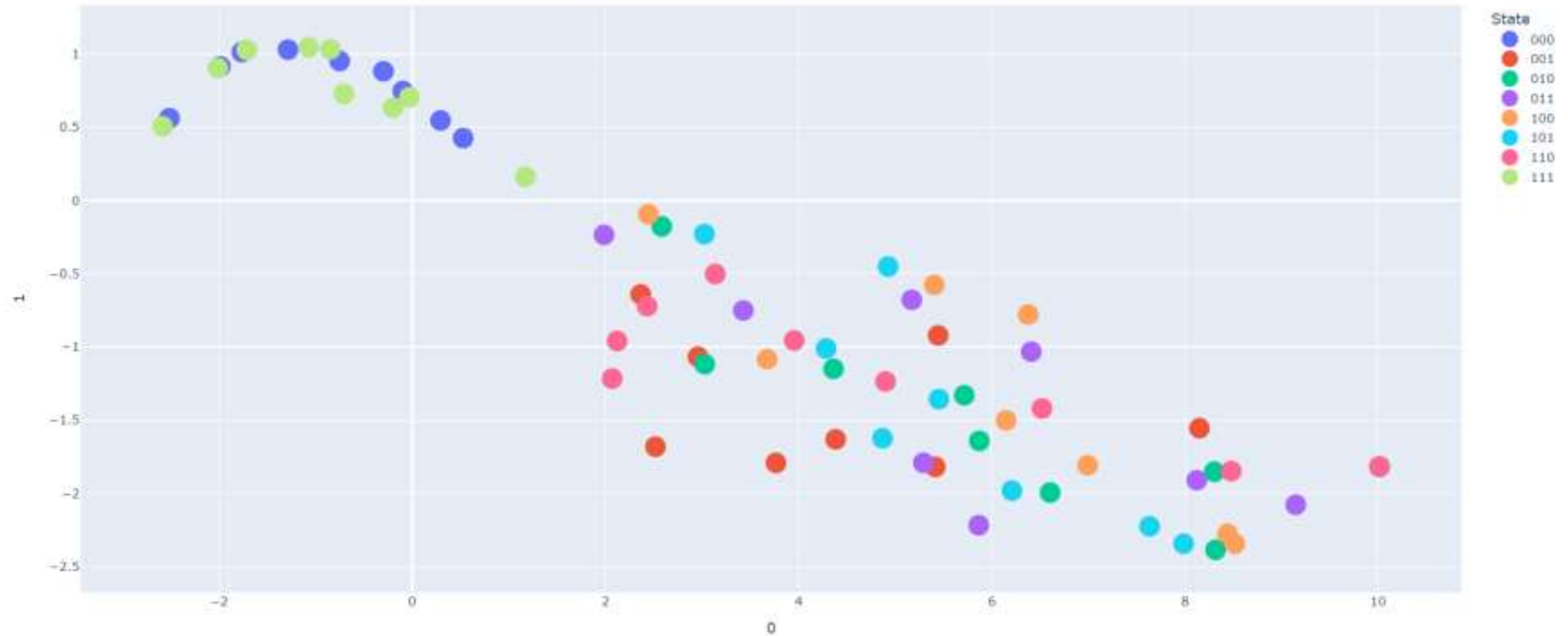
Question: What is T-SNE *actually* doing?

Data Visualization



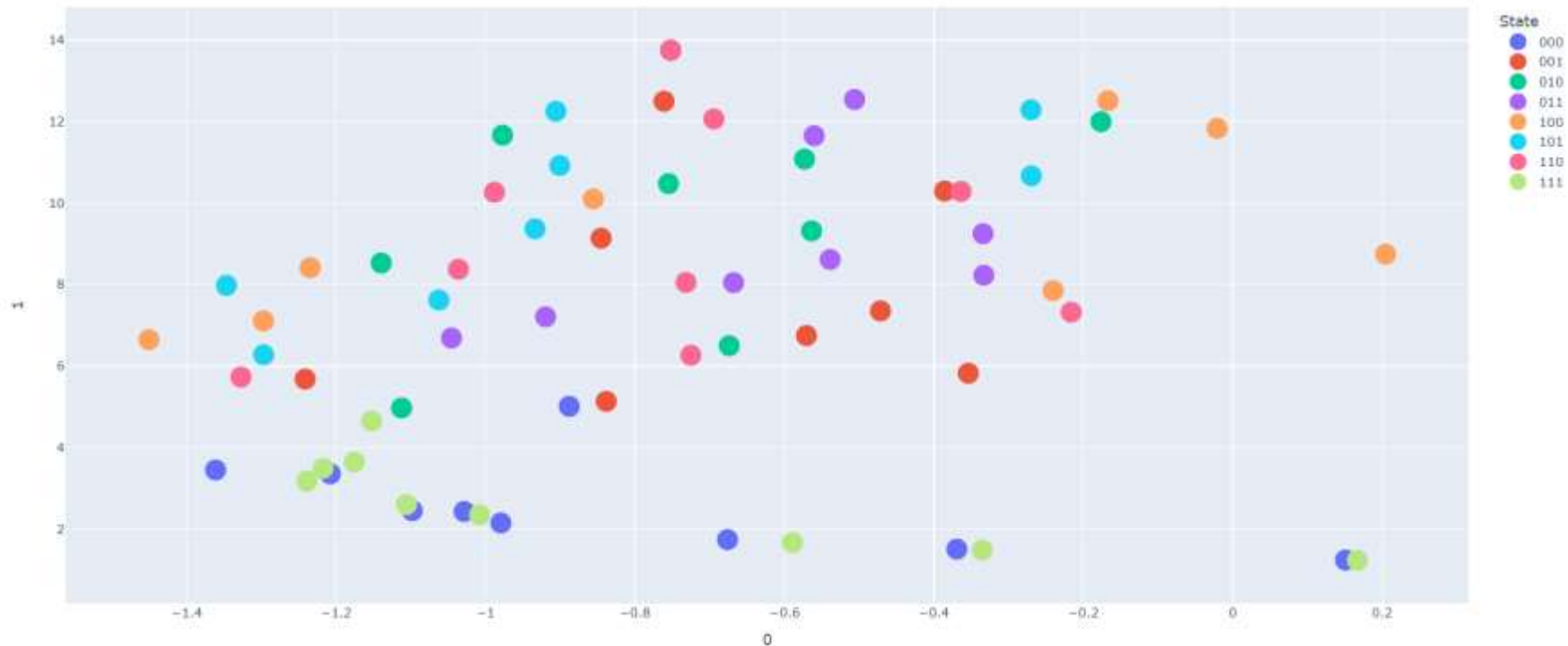
Data Visualization

T-SNE Reduction to 2D (Trial 1)



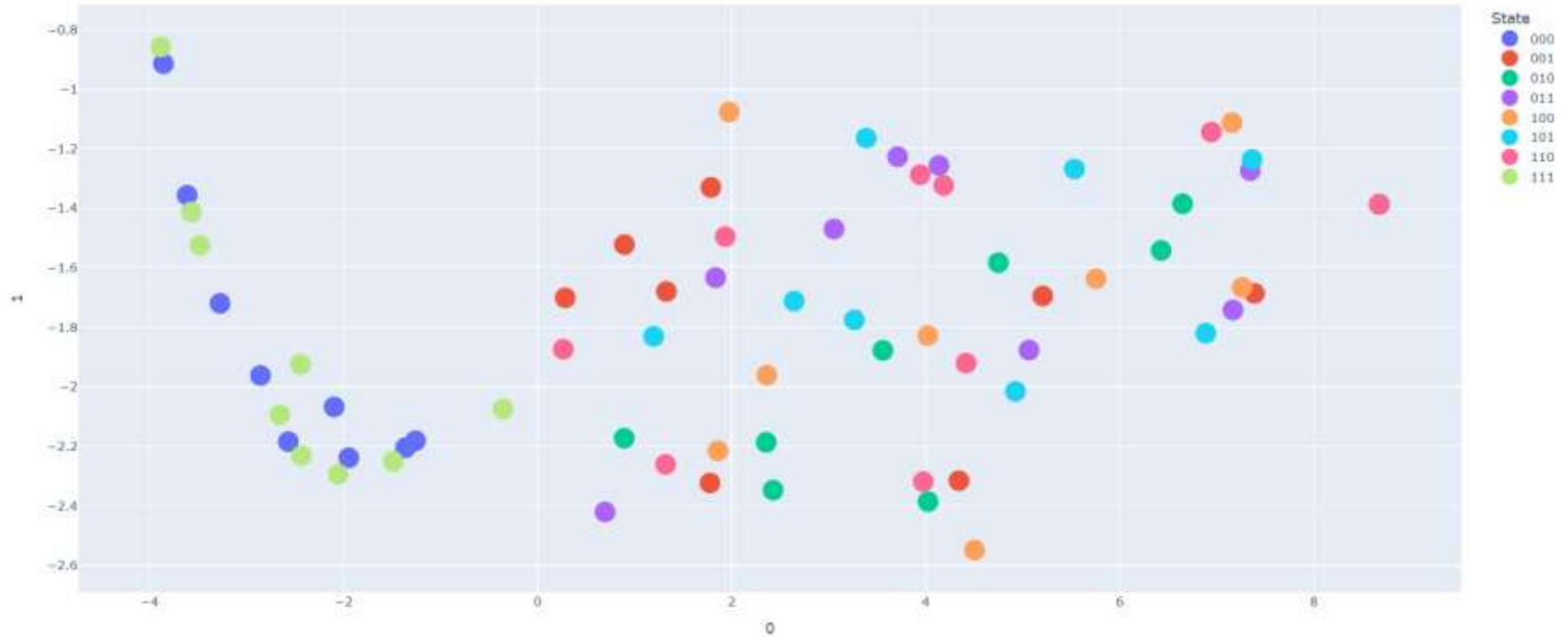
Data Visualization

T-SNE Reduction to 2D (Trial 2)



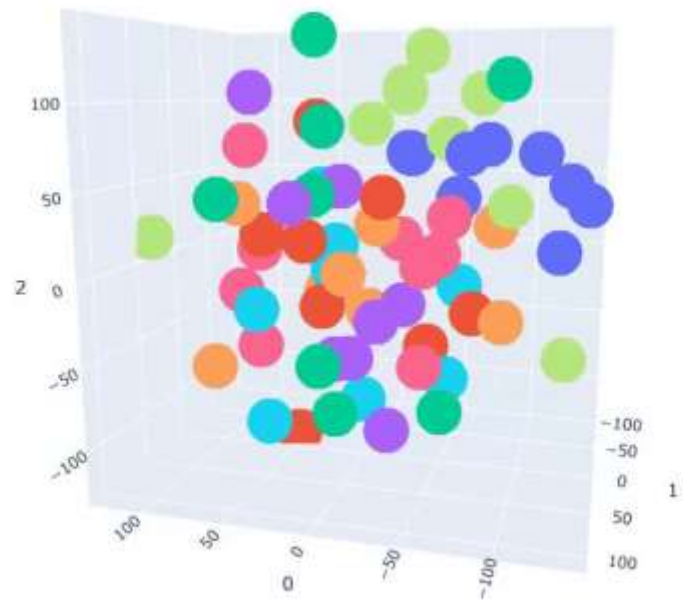
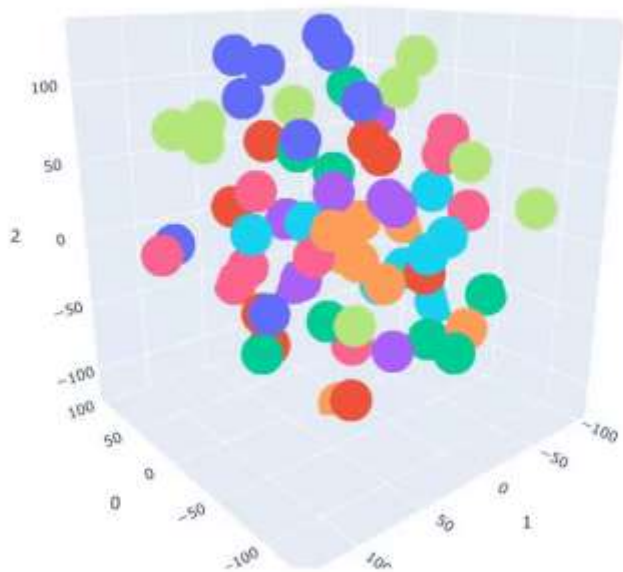
Data Visualization

T-SNE Reduction to 2D (Trial 3)



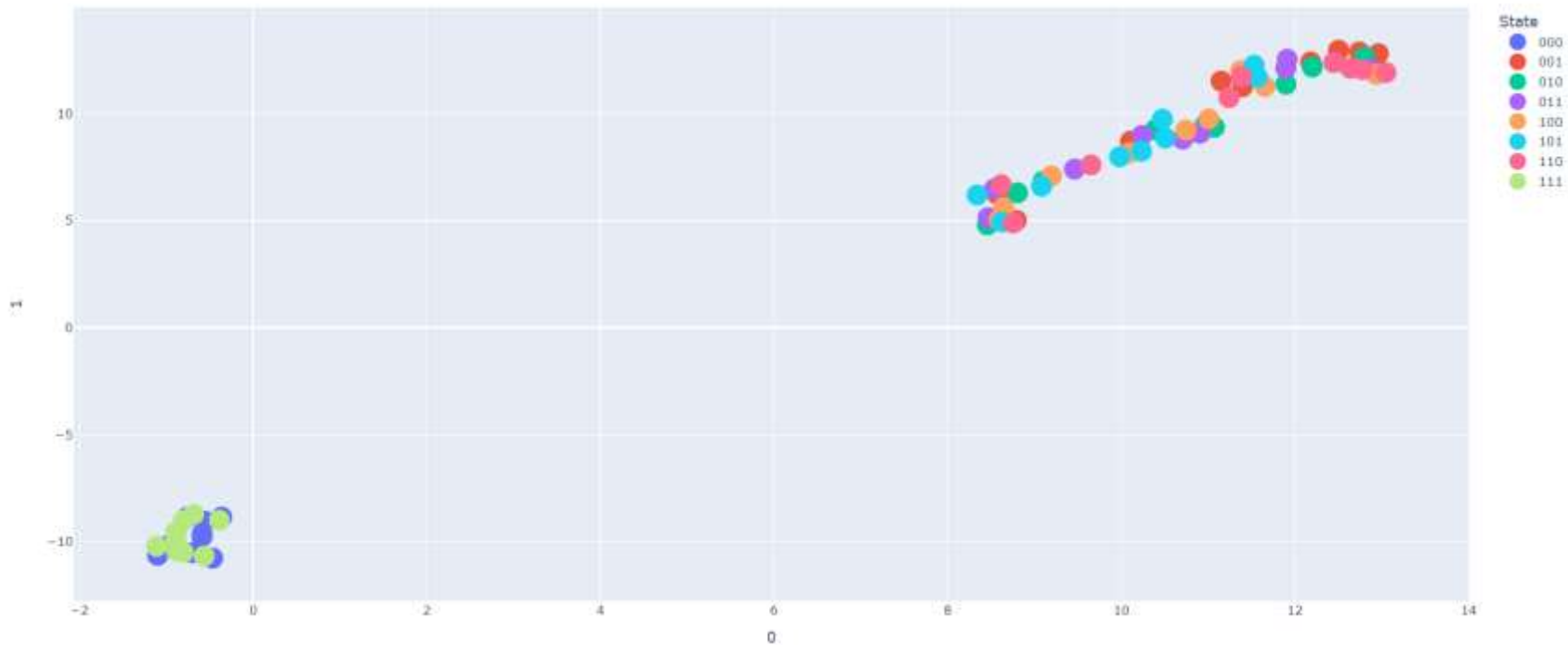
Data Visualization

T-SNE Reduction to 3D (Trial 4 and 5)



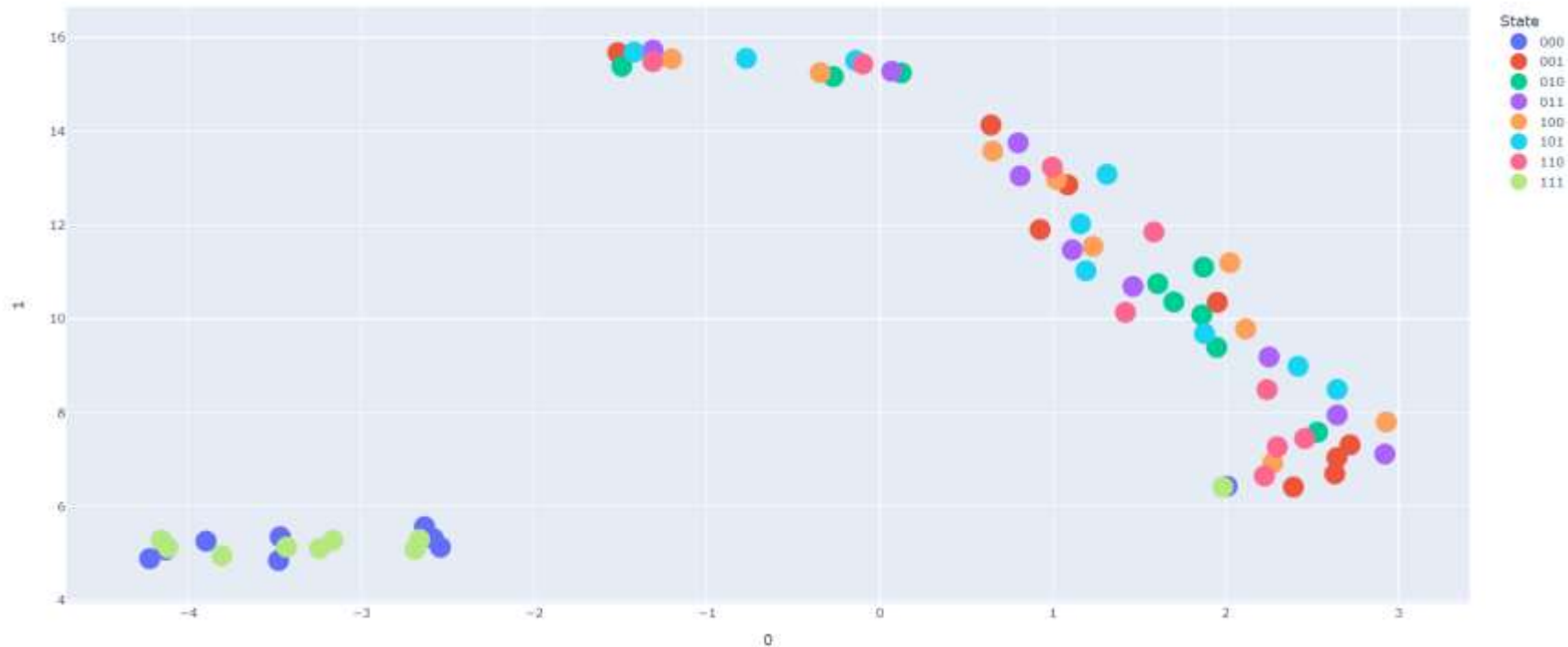
Data Visualization

UMAP Reduction to 2D (Trial 1)



Data Visualization

UMAP Reduction to 2D (Trial 2)



Data Visualization

UMAP Reduction to 3D (Trial 3 and 4)

