Grover's Search Algorithm

Abhay Saxena

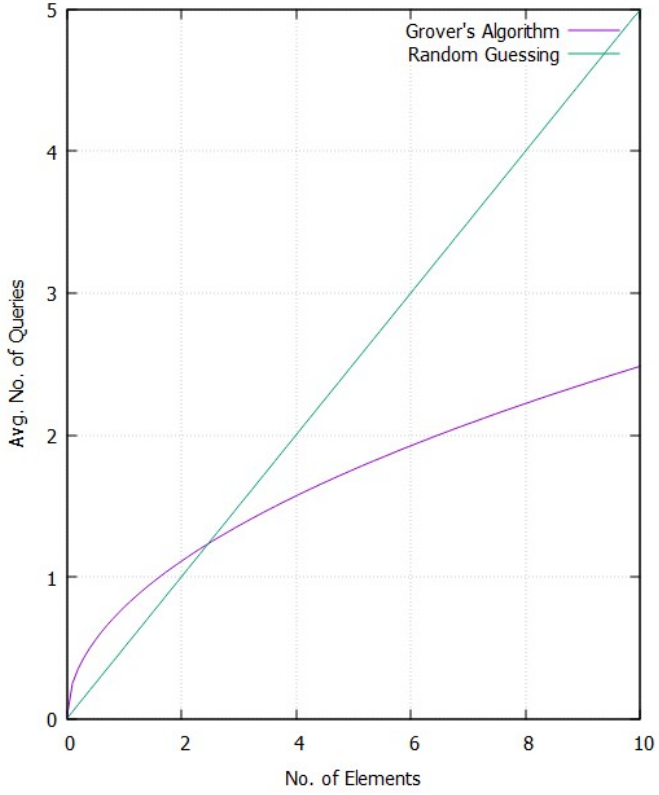
Problem

- Phone directory containing N names arranged in completely random order (unstructured data).
- Aim is to find a particular person's phone no.
- Best way to do it classically is to simply go though every single n^{th} entry until we reach the name being searched (average number of database queries grows linearly)
- However it could be done in less no. of steps using quantum computers

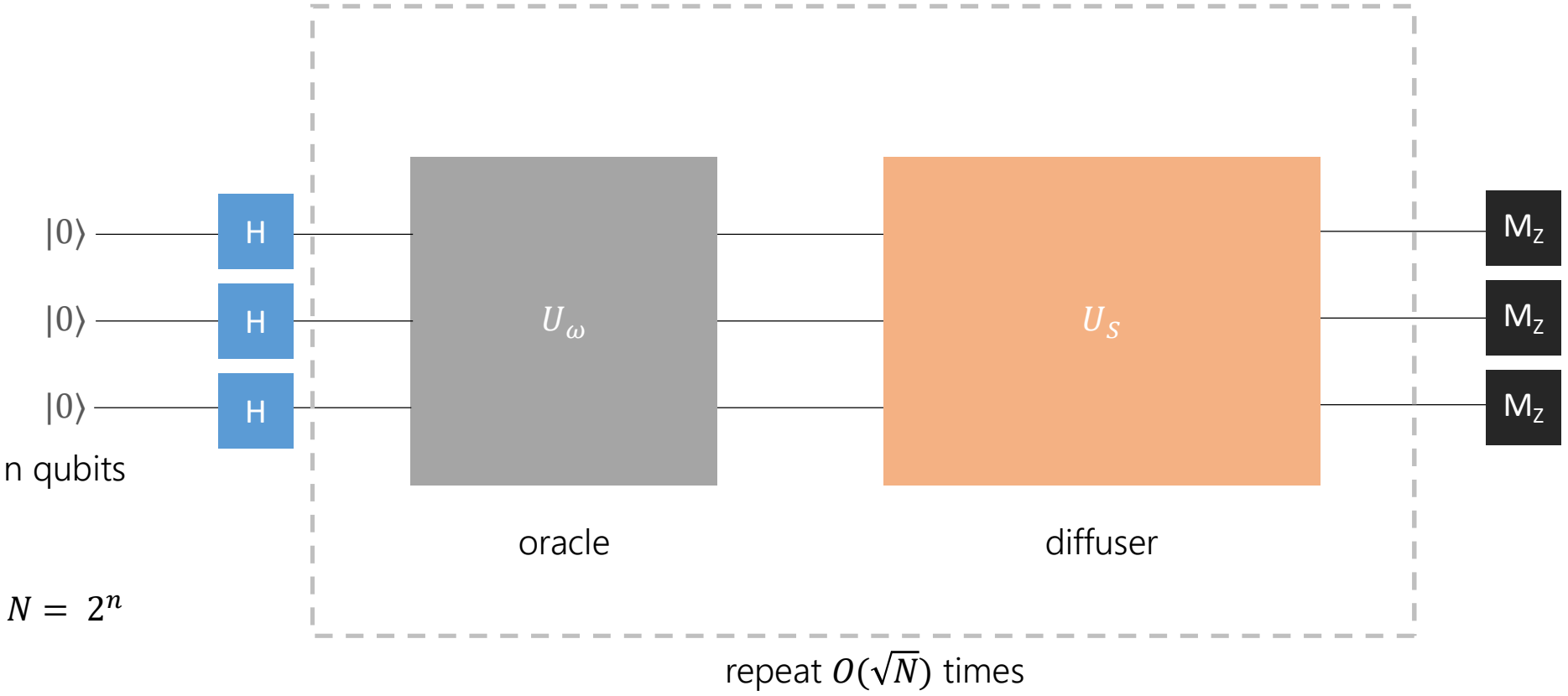
Name	No.
Sharad	12345678
Gowtham	87654321
Sagnik	12348765
Ananya	56784321
Pranav	56781234

Searching for a Needle in a Haystack

- Quantum mechanics can speed up a range of search applications over unsorted data.
- by having the input and output in superpositions of states, we can find an object in $O(\sqrt{N})$ (approx $\pi\sqrt{N}/4$) quantum mechanical steps instead of $O(N)$ (approx N/2) classical steps.



Basic structure



$N = 2^n$

Overview

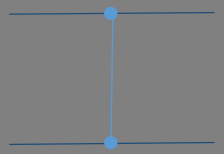
Ultimately we are trying to input all possible states in a superposition to this quantum circuit and trying to maximize the probability of getting output as out desired selected element(s).

- First we create an equal superposition of every possible input to the oracle. We can create this superposition by applying a H-gate to each qubit. We'll call this equal superposition state $|s\rangle$
- Then we run U_ω on this superpostition state. Oracle holds the table of elements and is used for querying data.
- Finally we apply diffuser operator which works along with oracle to magnify the amplitude of desired results

Oracle

- In order to simplify the database to a set of rules in bits for input and output we define certain blackboxes which input a possible
- it basically rotates the current state around perpendicular to $|\omega\rangle$.
- we can even use Toffoli gate to implement our desired oracle from classic oracles making its reversible version.

```
#just a simple oracle for ω = |11>
oracle = QuantumCircuit(2)
oracle.cz(0,1)  # not for |00>,|10>,|01>
oracle.draw()
```



$$U_\omega = I - 2|\omega\rangle\langle\omega|$$

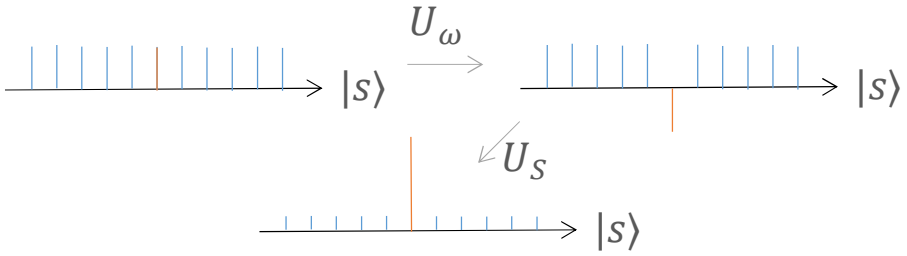
let $N = 2^n$ such that there are n qubits

$$|\omega\rangle = \frac{1}{\sqrt{no.\ of\ items\ in\ k}} \sum_{x \in k} |x\rangle$$

where k is set of desired outcomes

Diffuser

- In order to increase the magnitude of all the states with phase change due to oracle (i.e desired elements), we use diffuser which amplifies their probability and reduce the probability of others
- It basically rotates current state around the perpendicular to $|s\rangle$



$$U_S = 2|s\rangle\langle s| - I$$

$$|s\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle$$

$$U_s\sum_i a_i|i\rangle = \sum_i (2\langle a\rangle - a_i)|i\rangle$$

for arbitrary state

$$\langle\omega|s\rangle = \frac{\sqrt{k}}{\sqrt{N}}$$

$$R_g = U_S U_\omega$$

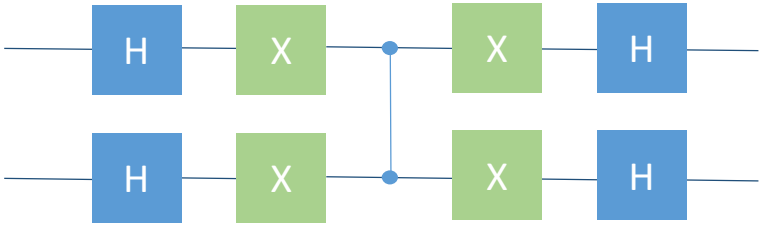
where R_g is Grover's Operator to be repeated $O(\sqrt{N})$ times

2 qubit diffuser

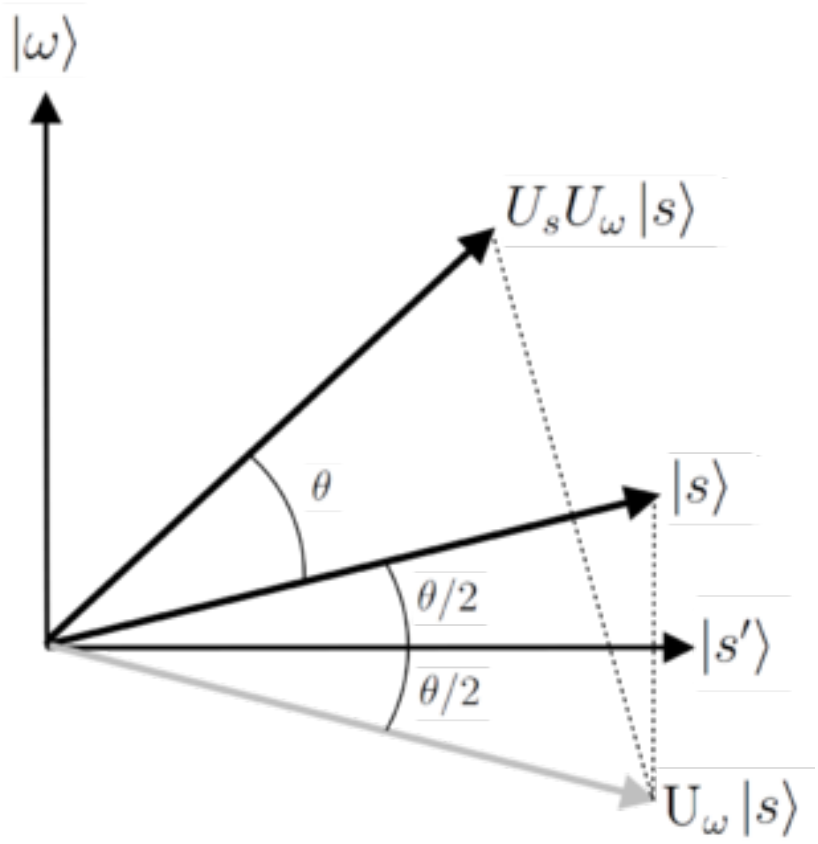
$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- to imitate the effect of reflection around the state $|s\rangle$, we simply create a transformation map $|s\rangle \rightarrow |11\rangle$ as we know an operator which reflects araound $|11\rangle$.
- so in short, we create $|s\rangle$ from $|00\rangle$ by h gates
- then do the transformation $|s\rangle \rightarrow |11\rangle$.
- reflect around $|11\rangle$ using cz.
- do the transformation $|11\rangle \rightarrow |s\rangle$.
- undo the H gate by applying it again.

```
diffuser.cz(0,1)
diffuser.x([0,1])
diffuser.h([0,1])
diffuser.draw()
grover = grover.compose(oracle)
grover = grover.compose(diffuser)
grover.measure_all()
```

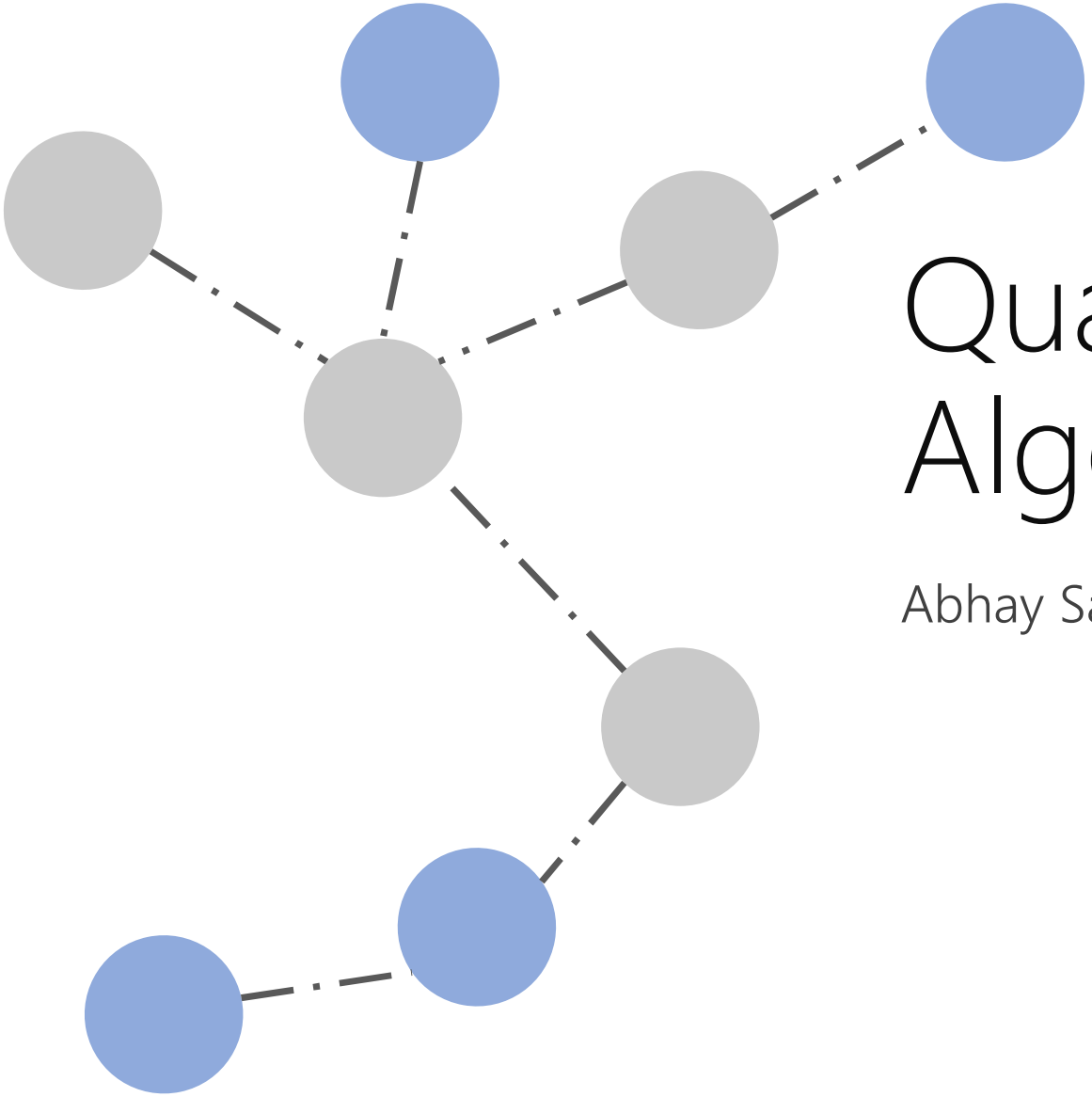


Geometric interpretation



Calculation no. of iterations

- $Angle\ btw.\ |s\rangle\ and\ |\omega\rangle = \frac{\pi}{2} - \frac{\theta}{2}$
- if we want to reach $|\omega\rangle$ in m iterations, we need $m \times\ \theta\ = \frac{\pi}{2} - \frac{\theta}{2}$
- $m = \frac{\pi}{2\theta} - \frac{1}{2}$
- $as\ |s\rangle = \frac{1}{\sqrt{N}}(\ |0\rangle\ +\ \ |1\rangle + \ldots +\ |\omega\rangle + \ldots |N-1\rangle)$ for the case that *even if there is just one solution state in* $|\omega\rangle$ *we can say,*
- $sin(\frac{\theta}{2}) = \frac{1}{\sqrt{N}}\ so,\ \frac{\theta}{2} \approx \frac{1}{\sqrt{N}}$ rad$(\approx \frac{\sqrt{k}}{\sqrt{N}}\ for\ k\ elements\ in\ |\omega\rangle)$
- $Hence,\ no.\ of\ iterations\ m \approx \frac{\pi}{4}\sqrt{N}$
- *interestingly no quantum Turig machine can do it in less than O(√N) iterations*



Quantum Algorithms

Abhay Saxena

classic: $O\left(\exp\left(\sqrt[3]{\frac{64}{9}n(\log n)^2}\right)\right)$

quantum: $O(n^2 \log n \log\log n)$

Shor's Algorithm

Modular Exponentiation Function

Quantum Fourier Transformation

GCD calculation and trial to find factors

Protocol

N=pq (find p and q, N is odd)

1. Pick a number ‘a’ that is coprime with N
2. Find the ‘order’ r of the function $a^r (mod\ N)$

$$\equiv smallest\ r\ st.\ a^r \equiv 1\ (mod\ N)$$

3. if r is even:

N divides: $a^r - 1 = (a^{r/2} - 1)(a^{r/2} + 1)$

$$x = a^{r/2}\ (mod\ N)$$

if x+1 ≢ 0 (mod N):

$$\{p,q\} \text{ contained in set } \{\gcd(x+1,N), \gcd(x-1,N)\}$$

polynomial complexly computable by classic computer

else: find another ‘a’

N=15

N=pq (find p and q (3 and 5), N is odd)

1. Pick a number ‘a’ that is coprime with 15 say a=7
2. Find the ‘order’ r of the function $7^r (mod\ 15)$

$$smallest\ r\ st.\ 7^4\ (mod\ 15) \equiv 1 : \mathrm{r} = 4$$

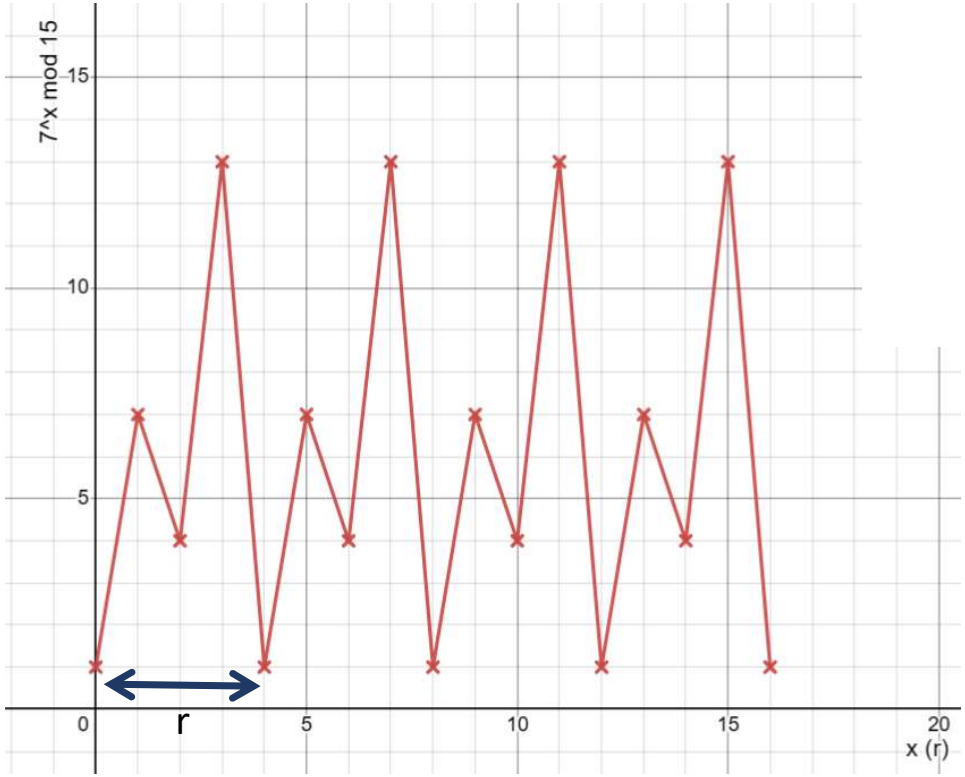
$$(\ 7^4 - 1)\ mod\ 15\ \equiv 0$$
$$(7^{4/2} - 1)(7^{4/2} + 1) \equiv 0$$

so 15 divides 48*50

i.e. {p,q} contained in set {gcd(x+1,N), gcd(x-1,N)}

Modular Exponentiation Function

So far the algorithm provides no advantage over classic method of simply finding the period as the problem is reduced to simple Modular Exponentiation Function and can be solved using discrete fourier Transformation. Thus, for quantum advantage we introduce Quantum Phase Estimation(QPE) using Quantum Fourier Transformation(QFT).



Quantum Fourier Transformation

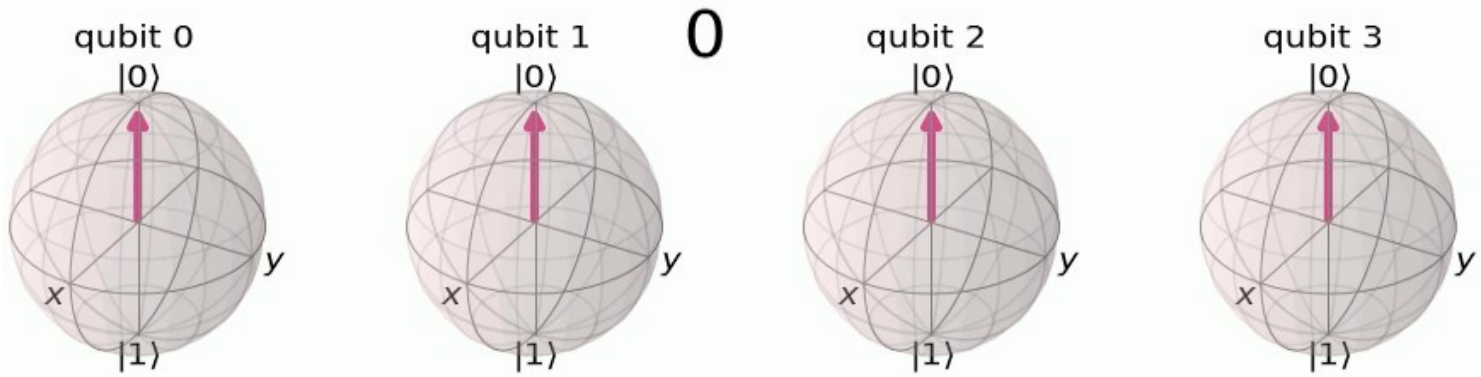
Descrete Fourier Transformation acts on a vector (x_0, x_1,...x_n) and maps it to the vector (x_0, x_1,...x_n)

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_N^{jk}$$

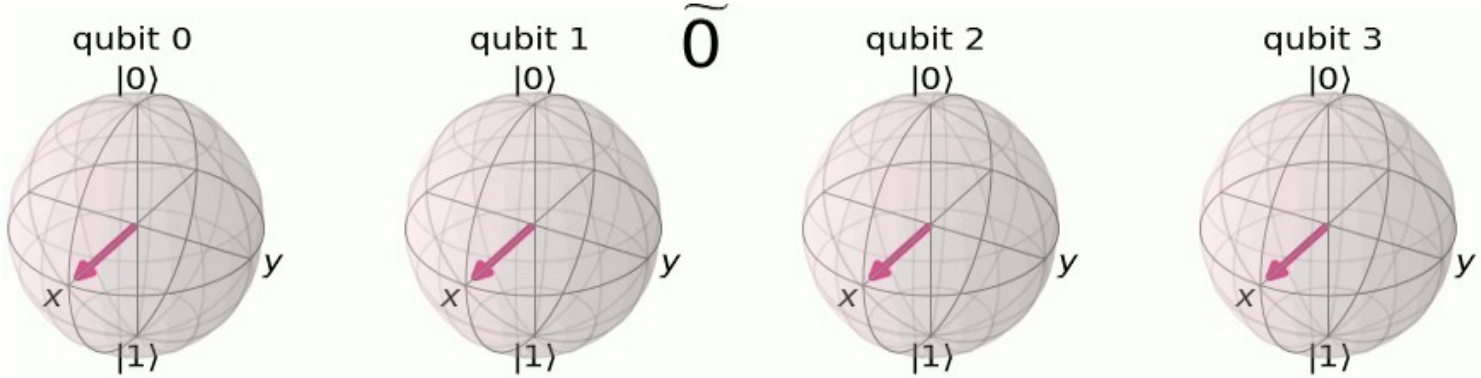
Quantum Fourier transform acts on a quantum state $|X\rangle = \sum_{j=0}^{N-1} x_j |j\rangle$ and maps it to $|Y\rangle = \sum_{k=0}^{N-1} y_k |k\rangle$ according to formula

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_N^{jk}$$

where $\omega_N^{jk} = e^{2\pi i \frac{jk}{N}}$



Computational Basis (in 0 and 1)



gif from qiskit textbook

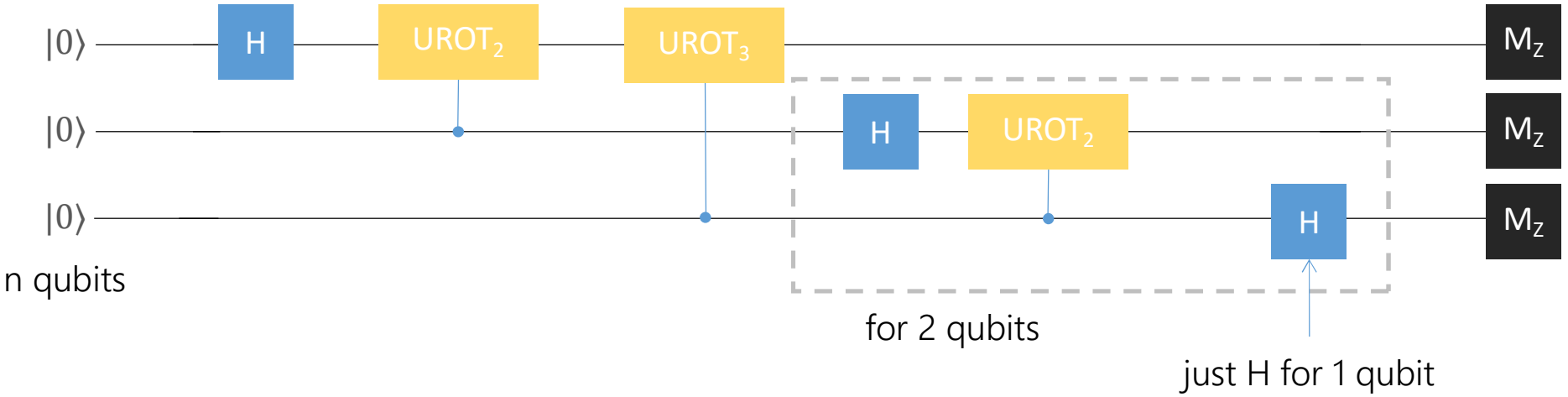
Fourier Basis (in 0 and 1)

$$for\ kth\ Fourier\ basis\ leftmost\ qubit\ is\ rotated\ by\ \frac{k}{2^n} \times 2\pi\ radians$$

Circuit for QFT

$$UROT_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix}$$

$$\omega_N^{jk} = e^{2\pi i \frac{jk}{N}}$$



$$|\tilde{x}\rangle = \frac{1}{\sqrt{N}}(|0\rangle + e^{\frac{2\pi i x}{2^1}}|1\rangle) \otimes \frac{1}{\sqrt{N}}(|0\rangle + e^{\frac{2\pi i x}{2^2}}|1\rangle) \ldots\ldots \otimes \frac{1}{\sqrt{N}}(|0\rangle + e^{\frac{2\pi i x}{2^n}}|1\rangle)$$

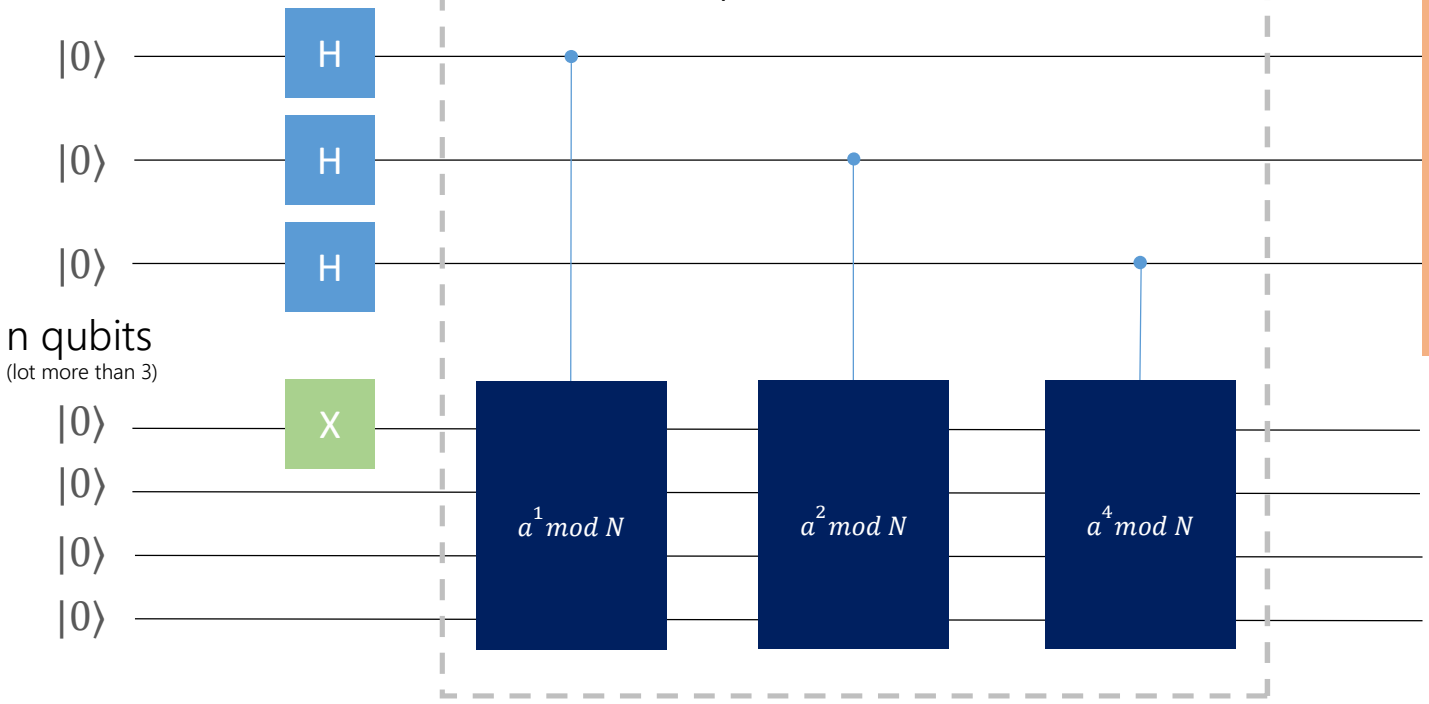
Circuit for QPE

the returned output divided by 2^n will give resp. fractions denominators of which should reresent the possible r's

$$U(\theta,\phi,\lambda) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\phi}\sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix}$$

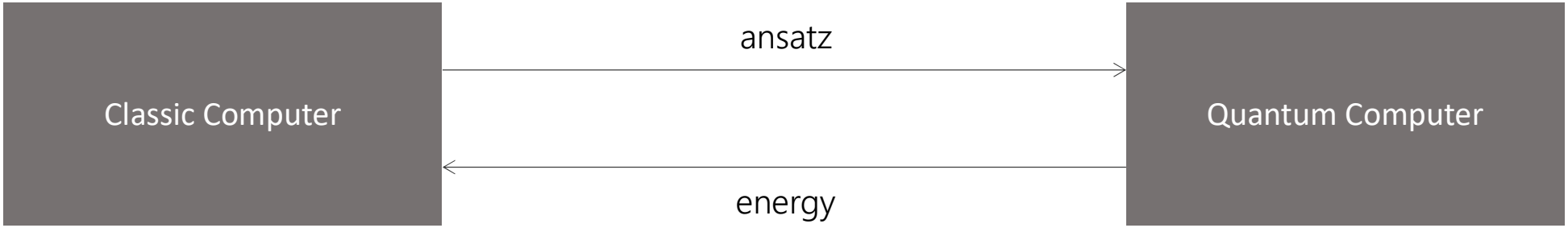
$$U|Y\rangle = |a\hat{}y \bmod N\rangle$$

phase kickback



$$|\tilde{x}\rangle = \left(\frac{1}{\sqrt{2}}\right)^n (|0\rangle + e^{i\theta_\psi 2^{n-1}}|1\rangle) \otimes (|0\rangle + e^{i\theta_\psi 2^{n-2}}|1\rangle) \ldots\ldots \otimes (|0\rangle + e^{i\theta_\psi 2^0}|1\rangle)$$

Variational Quantum Eigensolver



Variational Method

- we know that and eigenvector $|\psi_i\rangle$ of a matrix A, does not vary under transformation upto eigenvalue. $A|\psi_i\rangle = \lambda_i|\psi_i\rangle$.
- Eigenvalue of any Hermitian matrix has property $\lambda_i = \lambda_i^*$
- $H = \sum_{i=1}^{N} \lambda_i \, |\psi_i\rangle\langle\psi_i|$
- $\langle H\rangle_\psi = \langle\psi|H|\psi\rangle = \langle\psi|(\sum_{i=1}^{N} \lambda_i \, |\psi_i\rangle\langle\psi_i|)|\psi\rangle = \sum_{i=1}^{N} \lambda_i \, \langle\psi|\psi_i\rangle\langle\psi_i|\psi\rangle$

$$= \sum_{i=1}^{N} \lambda_i \, |\langle\psi|\psi_i\rangle|^2$$

so it is clear $\lambda_{min} \leq \langle H\rangle_\psi = \langle\psi|H|\psi\rangle = \sum_{i=1}^{N} \lambda_i \, |\langle\psi|\psi_i\rangle|^2$

this eqn is known as variational method for $\langle H\rangle_{\psi_{min}} = \lambda_{min}$

Variational Forms

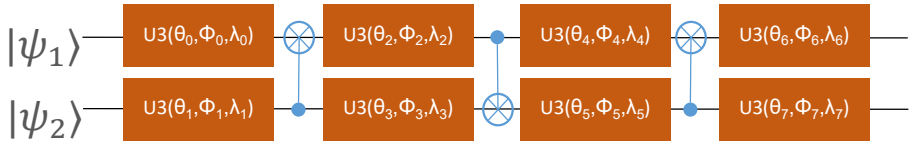
- we select some initial guess $|\psi\rangle$ (called ansatz) for $|\psi_{min}\rangle$, find it's expectation $\langle H\rangle_{\psi}$ to update to new guess.
- VQE varies these ansatz using parameterized circuit with a fixed form. Such Circuits are called Variational Forms.
- Let te action of Variational form be represented by linear transformation $U(\theta)$
- $U(\theta)|\psi\rangle = |\psi(\theta)\rangle$ is the optimized output state. Through iterative optimization, it aims to yield expectation close to $\langle H\rangle_{\psi_{min}} = \lambda_{min}$.
- Closeness of a state to E_{gs} is measured through manhattan distance.
- n Qubit variational form would be able to generate any possible state $|\psi\rangle$

where $|\psi\rangle \in C^N \; and \; N = 2^n$.

Variational forms

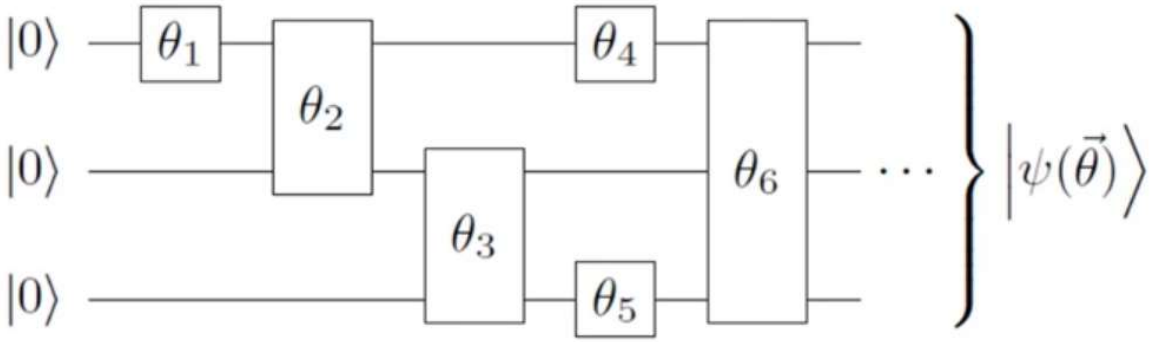
- for n=1, U3 gate is a variational form capable of generating any possible state
- for n=2, where two body Interactions happen, entanglement, must be considered to achieve universality.

$$U3(\theta,\phi,\lambda) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda}\sin(\frac{\theta}{2}) \\ e^{i\phi}\sin(\frac{\theta}{2}) & e^{i\lambda+i\phi}\cos(\frac{\theta}{2}) \end{pmatrix}$$



NOTE:during the optimization process, the variational form does not limit the set of attainable states over which the expectation value of Hamiltonian.This ensures that the minimum expectation value is limited only by the capabilities of the classical optimizer.

General parametrized state



Chosing Variational forms

- quantum hardware has various types of noise and so objective function evaluation (energy calculation) may not necessarily reflect the true objective function.
- Appropriate optimizer should be selected by considering the requirements of an application.
- gradient descent: each parameter is updated in the direction yielding the largest local change in energy (often gets stuck at poor local optima, high number of circuit evaluations)
- Simultaneous Perturbation Stochastic Approximation optimizer (SPSA): approximates the gradient of the objective function with only two measurements. Concurrently perturbing all the parameters in a random fashion.
- If no noise is present (Perfect simulator for VQE):
  - Sequential Least Squares Programming optimizer (SLSQP): use if objective function and the constraints are twice continuously differentiable.
  - Constrained Optimization by Linear Approximation optimizer (COBYLA): only performs one objective function evaluation per optimization iteration (and thus the number of evaluations is independent of the parameter set's cardinality).

Gradient Descent for 1 qubit sustem

```
import numpy as np
np.random.seed(999999)
p0 = np.random.random()
target_distr = {0: p0, 1: 1 - p0}
```

```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit.circuit import Parameter

p = [Parameter("theta"), Parameter("phi"), Parameter("lam")

def form(p):
    qr = QuantumRegister(1, name="q")
    cr = ClassicalRegister(1, name="c")
    qc = QuantumCircuit(qr, cr)
    qc.u(p[0], p[1], p[2], qr[0])
    qc.measure(qr, cr[0])
    return qc

qc = form(p)
```

```
Parameters Found: [ 1.47924356 -0.29323942  2.00245954]
Target Distribution: {0: 0.308979188922057, 1: 0.691020810779 43}
Obtained Distribution: {1: 0.7119140625, 0: 0.2880859375}
Cost: 0.039833778441140 04
```

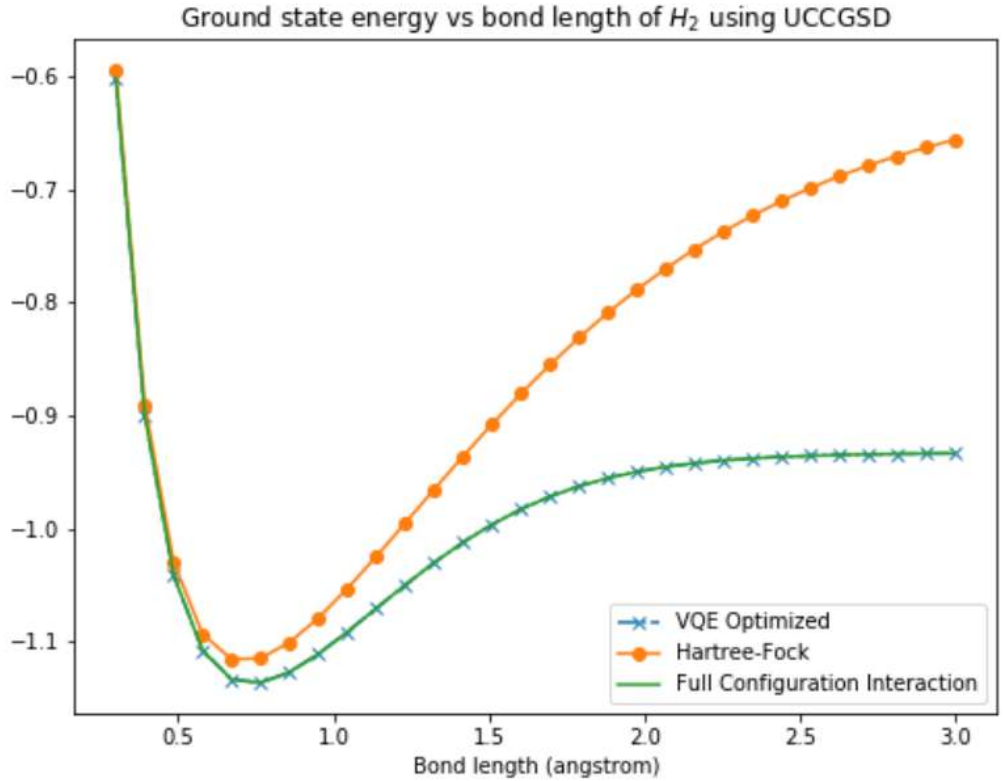
```
from qiskit_aer.primitives import Sampler, Estimator

sampler = Sampler()
def objective_function(p):
    result = sampler.run(circuits=qc, parameter_values=p).result()
    output_distr = result.quasi_dists[0]
    cost = sum(
        abs(target_distr.get(i, 0) - output_distr.get(i, 0))
        for i in range(2**qc.num_qubits)
    )
    return cost
```

```
from qiskit.algorithms.optimizers import SPSA, SLSQP, COBYLA
optimizer = COBYLA(maxiter=500, tol=0.0001)
initial_point = np.random.rand(3)
result = optimizer.minimize(fun=objective_function, x0=initial_point)
output_distr = (
    sampler.run(circuits=qc, parameter_values=result.x).result().quasi_dists[0]
)
print("Parameters:", result.x)
print("Target DIST:", target_distr)
print("Obtained DIST:", output_distr)
print("Cost:", objective_function(result.x))
```

Brief summary as to how to make VQE work

1. Map the problem that you want to solve to finding the ground state energy of a Hamiltonian (ie, a molecule problem or some cost function)
2. Prepare a trial state with some collection of parameters
3. Run that through the parameterized quantum circuit
4. Measure expectation values of Hamiltonian (by measuring each of the Pauli strings or the 'observables' of the Hamiltonian)
5. Calculate the 'energy' corresponding to the trial state by summing up all the measurements in step #4
6. Update all the parameters via some optimization algorithm (ie, some form of gradient descent)
7. Now you have the first iteration of the trial state with some new parameters to feedback into step 2 and just repeat everything all over again until you get the lowest possible energy.

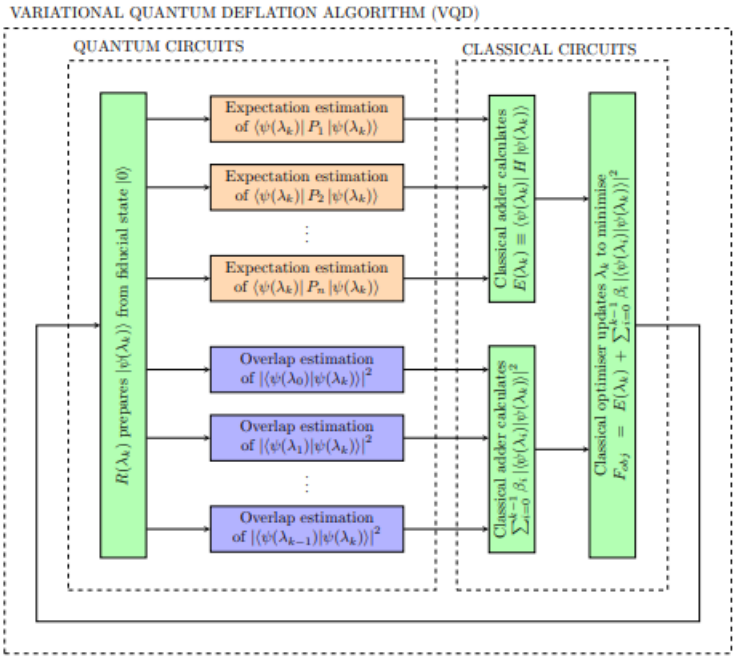
$$\epsilon_i \psi_i(\mathbf{r}) = \left(-\frac{1}{2}\nabla^2 + V_{ion}(\mathbf{r})\right)\psi_i(\mathbf{r}) + \sum_j \int d\mathbf{r}' \frac{|\psi_j(\mathbf{r}')|^2}{|\mathbf{r}-\mathbf{r}'|}\psi_i(\mathbf{r}) - \sum_j \delta_{\sigma_i\sigma_j} \int d\mathbf{r}' \frac{\psi_j^*(\mathbf{r}')\psi_i(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}\psi_j(\mathbf{r}) \quad .$$



Topics Covered

- Quantum Circuit Basics with Qiskit
- Postulates of QM
- Qubits
- Quantum Gates
- No-Cloning Theorem
- Quantum Teleportation
- QFT, QPE
- Super Dense Coding
- Density Matrix
- Measurement Postulates
- Grover’s Search Algorithm
- Shor’s Algorithm
- VQE

vqd



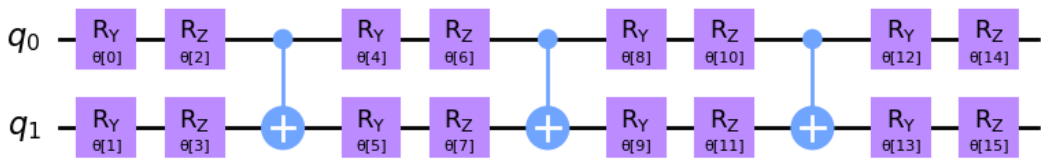
[1]:
```
import qiskit_nature
from qiskit_nature.second_q.formats.molecule_info import MoleculeInfo
from qiskit_nature.second_q.drivers import Psi4Driver
from qiskit_nature.second_q.transformers import FreezeCoreTransformer
from qiskit_nature.second_q.mappers import ParityMapper
qiskit_nature.settings.use_pauli_sum_op = False

mol = MoleculeInfo(
    # Coordinates in Angstrom
    symbols=["H", "H"],
    coords=([0.0, 0.0, -0.3625], [0.0, 0.0, 0.3625]),
    multiplicity=1,
    charge=0,
)

prob_unmod = Psi4Driver.from_molecule(mol).run()
```

```
problem = FreezeCoreTransformer(
        freeze_core=True, remove_orbitals=[-3, -2]
    ).transform(prob_unmod)

num_spatial_orbitals = problem.num_spatial_orbitals
num_particles = problem.num_particles

mapper = ParityMapper(num_particles=num_particles)  # Set Mapper

hamiltonian = mapper.map(problem.second_q_ops()[0])  # Set Hamiltonian

print(hamiltonian)
```

```
SparsePauliOp(['II', 'IZ', 'ZI', 'ZZ', 'XX'],
              coeffs=[-1.05016043+0.j,  0.40421466+0.j, -0.40421466+0.j,
-0.01134688+0.j,
  0.18037525+0.j])
```

[2]:
```
from qiskit.circuit.library import EfficientSU2
#ansatz = TwoLocal(rotation_blocks=['ry', 'rz'], entanglement_blocks='cz')
ansatz = EfficientSU2(hamiltonian.num_qubits)

from qiskit.algorithms.optimizers import SLSQP
optimizer = SLSQP(maxiter=150, max_evals_grouped=1)

ansatz.decompose().draw('mpl')
```

[2]:



[3]:
```
from qiskit.primitives import Sampler, Estimator
from qiskit.algorithms.state_fidelities import ComputeUncompute

estimator = Estimator()
sampler = Sampler()
fidelity = ComputeUncompute(sampler)
```

ComputeUnCompute uses the sampler primitive to calculate the state fidelity of two quantum circuits following the compute-uncompute method

fedility (state overlap): $|\langle\psi(x)|\phi(y)\rangle|^2$

The initial release of Qiskit Runtime includes two primitives:

Sampler: Generates quasi-probability distribution from input circuits.

Estimator: Calculates expectation values from input circuits and observables.

[4]:
```
k = 4
betas = [33,33,33,33]
```

[5]:
```
counts = []
values = []
steps = []

def callback(eval_count, params, value, meta, step):
    counts.append(eval_count)
    values.append(value)
    steps.append(step)
```

$$F(\lambda_k) \coloneqq \langle\psi(\lambda_k)|\, H\, |\psi(\lambda_k)\rangle + \sum_{i=0}^{k-1} \beta_i \left|\langle\psi(\lambda_k)|\psi(\lambda_i)\rangle\right|^2,$$

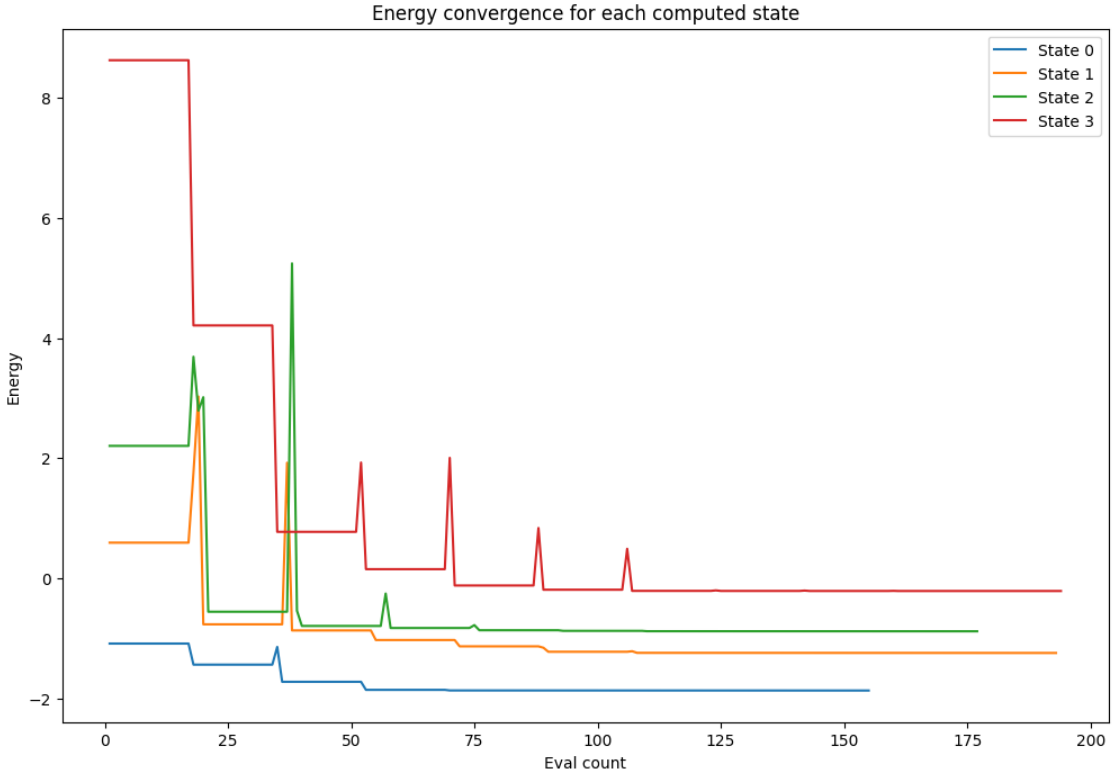
[6]:
```
from qiskit.algorithms.eigensolvers import VQD


vqd = VQD(estimator, fidelity, ansatz, optimizer, k=k, betas=betas,␣
 ↪callback=callback)
result = vqd.compute_eigenvalues(operator = hamiltonian)
vqd_values = result.eigenvalues
```

[7]:
```
print(vqd_values.real)
```

```
[-1.86712097 -1.24188254 -0.88113169 -0.21050596]
```

[8]:
```
import numpy as np
import matplotlib.pyplot as plt
g1=plt

g1.rcParams["figure.figsize"] = (12, 8)

steps = np.asarray(steps)
counts = np.asarray(counts)
values = np.asarray(values)

for i in range(1,5):
    _counts = counts[np.where(steps == i)]
    _values = values[np.where(steps == i)]
    g1.plot(_counts, _values, label=f"State {i-1}")
```

```
g1.xlabel("Eval count")
g1.ylabel("Energy")
g1.title("Energy convergence for each computed state")
g1.legend(loc="upper right")
g1.show()
```



[9]:
```
from qiskit.algorithms.eigensolvers import NumPyEigensolver


exact_solver = NumPyEigensolver(k=4)
exact_result = exact_solver.compute_eigenvalues(hamiltonian)
ref_values = exact_result.eigenvalues
```

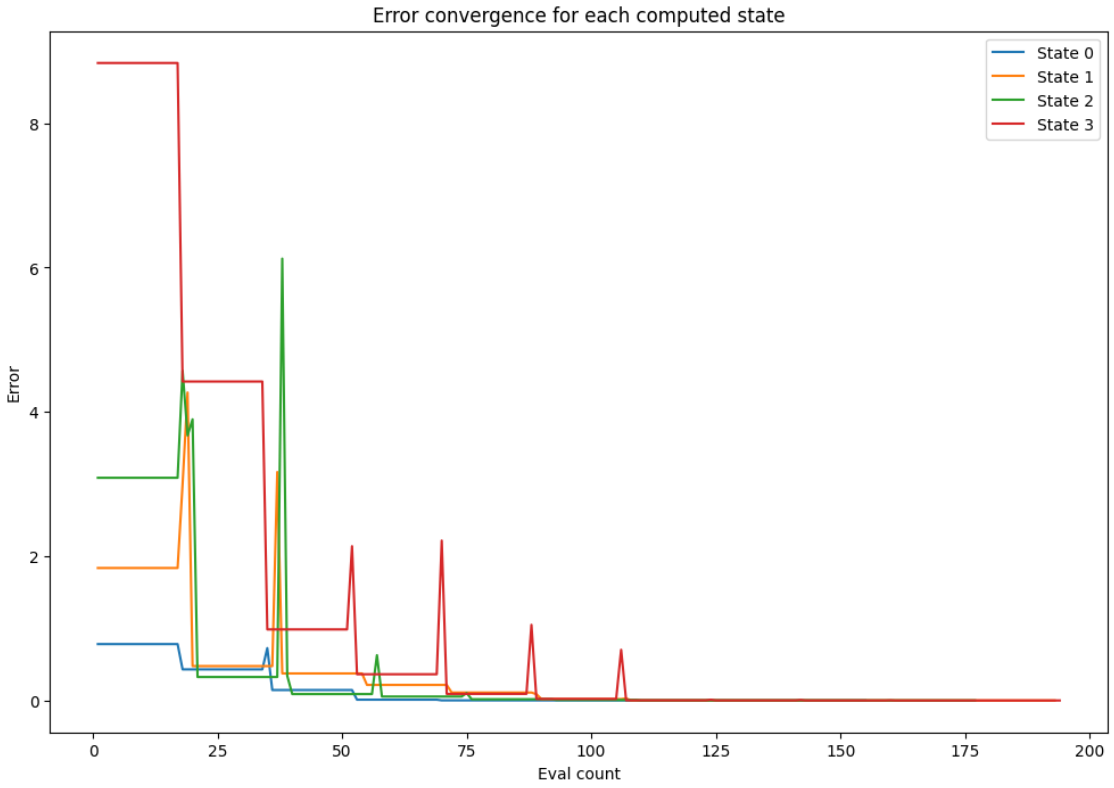
[10]:
```
print(f"Reference values: {ref_values}")
print(f"VQD values: {vqd_values.real}")
```

```
Reference values: [-1.86712098 -1.24188257 -0.88113207 -0.21050612]
VQD values: [-1.86712097 -1.24188254 -0.88113169 -0.21050596]
```

[11]:
```
print('errors in final VQD values (x 10^-8): ')
for i in range(0,4):
    print('state', i, ':', (abs(vqd_values.real-ref_values) * 10**8)[i])
```

```
errors in final VQD values (x 10^-8):
state 0 : 1.2845639441039225
state 1 : 2.069316917818753
state 2 : 37.753599502199364
state 3 : 16.311304948390948
```

[12]:
```python
import numpy as np
import matplotlib.pyplot as plt

g2 = plt
g2.rcParams["figure.figsize"] = (12, 8)

steps = np.asarray(steps)
counts = np.asarray(counts)
values = np.asarray(values)

for i in range(1,5):
    _counts2 = counts[np.where(steps == i)]
    _values2 = -(-values[np.where(steps == i)] + ref_values[i-1])
    g2.plot(_counts2, _values2, label=f"State {i-1}")

g2.xlabel("Eval count")
g2.ylabel("Error")
g2.title("Error convergence for each computed state")
g2.legend(loc="upper right")
g2.show()
```


Error convergence for each computed state
State 0
State 1
State 2
State 3
Error
Eval count
0
2
4
6
8
0
25
50
75
100
125
150
175
200

pvqd

Definition:

A time operator for small step

$$\delta t \in \mathbf{R} \text{ is } e^{-i\hat{H}\delta t}$$

$|\psi_{\omega(t)}\rangle$:let this describe quantum state of the system for time t.

p-VQD problem for d^{ω} and dt is thus-

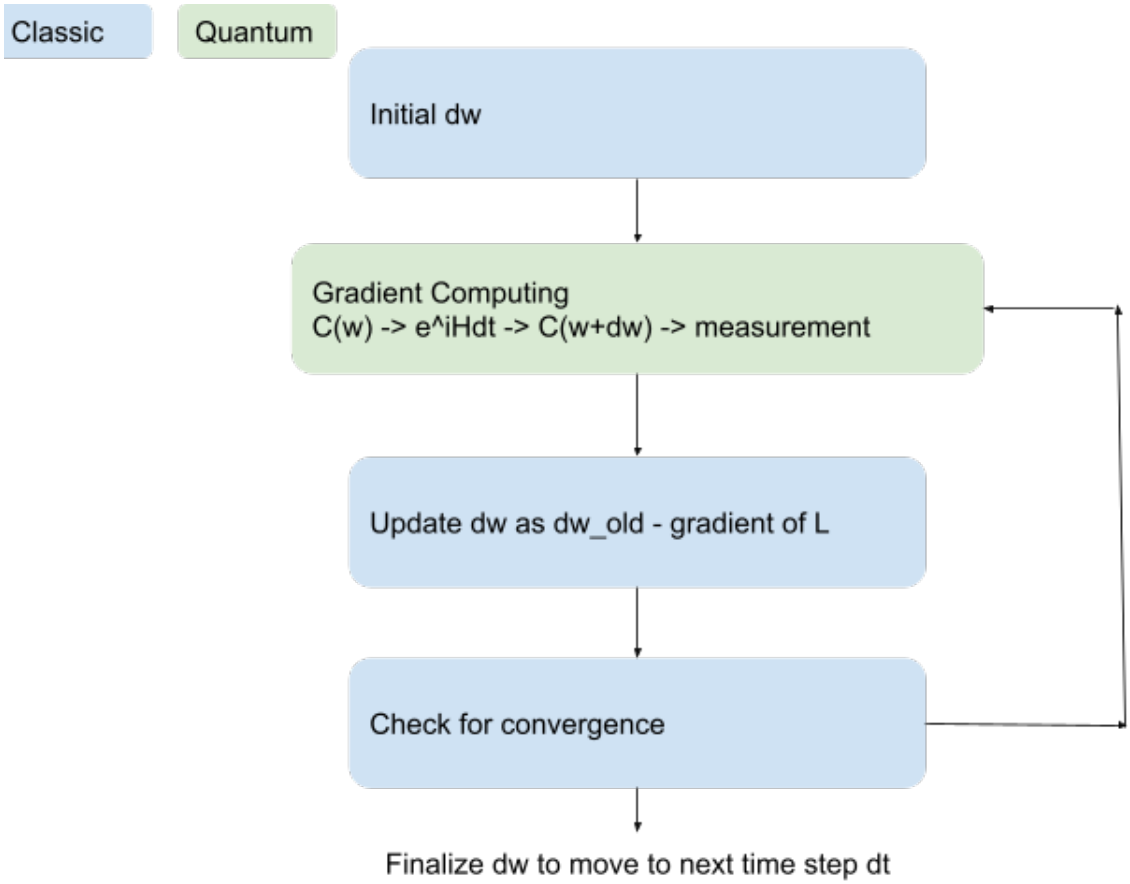
$$\arg\max_{dw \in \mathbb{R}^p} \left|\langle \phi(\delta t)|\psi_{w+dw}\rangle\right|^2$$

Optimally best next step to find the parameters is to minimise step-infidelity defined by:

$$L(dw, \delta t) = \frac{1 - |\langle 0|C^{\dagger}(w)e^{i\hat{H}\delta t}C(w+dw)|0\rangle|^2}{\delta t^2}$$

Where, $|\psi_w\rangle = C(w)|0\rangle$

Basically, this method projects the time step onto a variational form (ansatz) and uses a Trotter formula (given by the evolution argument) to calculate the next state for each timestep. By applying a traditional optimization procedure, the projection is found by maximising the fidelity of the Trotter-evolved state and the ansatz.



Basic Structure:

[6]:
```
import numpy as np

from qiskit.algorithms.state_fidelities import ComputeUncompute
from qiskit.algorithms.time_evolvers import TimeEvolutionProblem, PVQD
from qiskit.primitives import Estimator, Sampler
from qiskit.circuit.library import EfficientSU2
from qiskit.quantum_info import SparsePauliOp, Pauli
from qiskit.algorithms.optimizers import L_BFGS_B
```

[7]:
```
sampler = Sampler()
fidelity = ComputeUncompute(sampler)
estimator = Estimator()
hamiltonian = 0.1 * SparsePauliOp(["ZZ", "IX", "XI"])
observable = Pauli("ZZ")
ansatz = EfficientSU2(2, reps=1)
initial_parameters = np.zeros(ansatz.num_parameters)
```

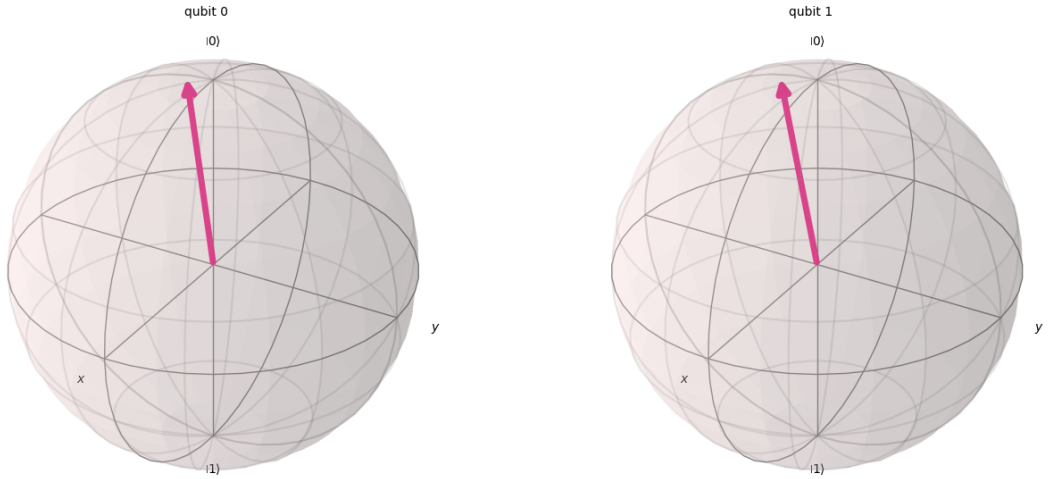
[8]:
```
time = 1
optimizer = L_BFGS_B()

# setup the algorithm
pvqd = PVQD(
```

```
    fidelity,
    ansatz,
    initial_parameters,
    estimator,
    num_timesteps=100,
    optimizer=optimizer,
)
```

[9]:
```
# specify the evolution problem
problem = TimeEvolutionProblem(
    hamiltonian, time, aux_operators=[hamiltonian, observable]
)

# and evolve!
result = pvqd.evolve(problem)
```

[10]:
```
from qiskit.visualization import plot_bloch_multivector
from qiskit.quantum_info import Statevector
quantum_state = Statevector.from_instruction(result.evolved_state)
plot_bloch_multivector(quantum_state)
```

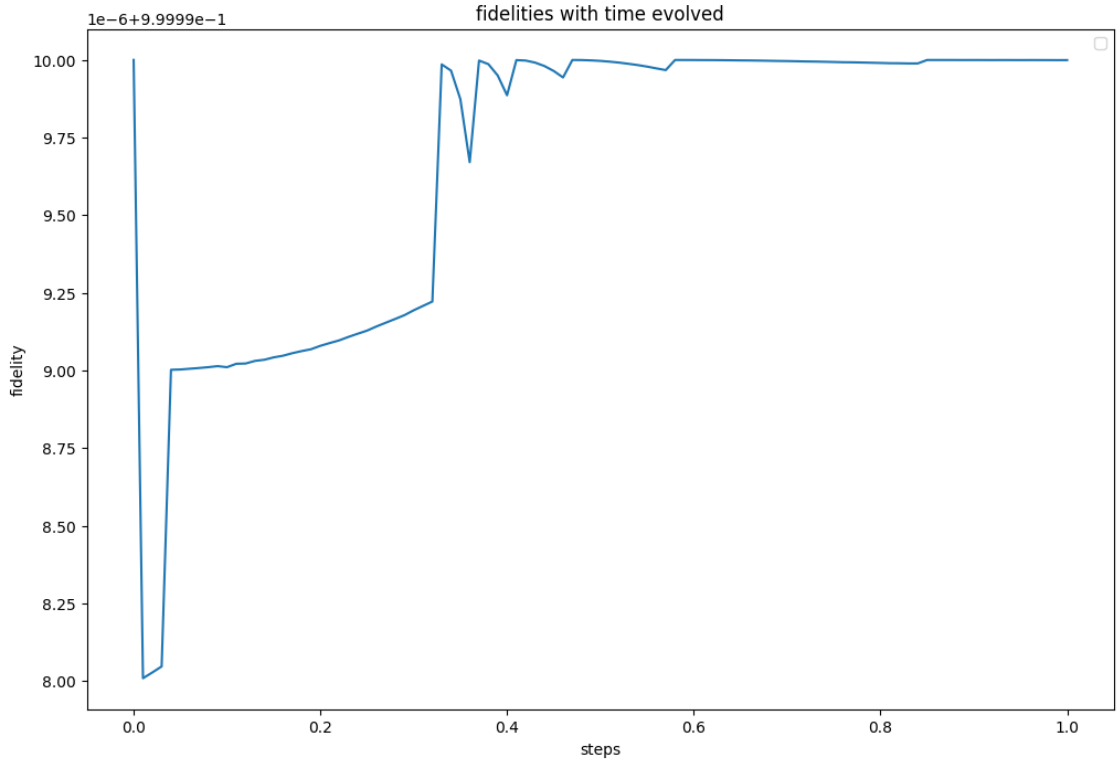
[10]:



[19]:
```
import numpy as np
import matplotlib.pyplot as plt
g1=plt

g1.rcParams["figure.figsize"] = (12, 8)


_counts = result.fidelities
```

```
_values = result.times
g1.plot(_values, _counts)
g1.xlabel("steps")
g1.ylabel("fidelity")
g1.title("fidelities with time evolved")
g1.legend(loc="upper right")
g1.show()
```

```
No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
```



ComputeUnCompute uses the sampler primitive to calculate the state fidelity of two quantum circuits following the compute-uncompute method

fedility (state overlap):

The initial release of Qiskit Runtime includes two primitives:

Sampler: Generates quasi-probability distribution from input circuits.

Estimator: Calculates expectation values from input circuits and observables.

[17]:
```
print( result.times)
```

```
[0.0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11, 0.12,
0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2, 0.21, 0.22, 0.23, 0.24, 0.25,
```

0.26, 0.27, 0.28, 0.29, 0.3, 0.31, 0.32, 0.33, 0.34, 0.35000000000000003, 0.36,
0.37, 0.38, 0.39, 0.4, 0.41000000000000003, 0.42, 0.43, 0.44, 0.45, 0.46,
0.47000000000000003, 0.48, 0.49, 0.5, 0.51, 0.52, 0.53, 0.54, 0.55, 0.56,
0.5700000000000001, 0.58, 0.59, 0.6, 0.61, 0.62, 0.63, 0.64, 0.65, 0.66, 0.67,
0.68, 0.6900000000000001, 0.7000000000000001, 0.71, 0.72, 0.73, 0.74, 0.75,
0.76, 0.77, 0.78, 0.79, 0.8, 0.81, 0.8200000000000001, 0.8300000000000001, 0.84,
0.85, 0.86, 0.87, 0.88, 0.89, 0.9, 0.91, 0.92, 0.93, 0.9400000000000001,
0.9500000000000001, 0.96, 0.97, 0.98, 0.99, 1.0]