Lesson 4: SQD application

In this lesson, we will apply SQD to estimate the ground state energy of a molecule.

In particular, we will discuss the following topics using the 4-step Qiskit pattern approach:

1. Step 1: Map problem to quantum circuits and operators
   - Setup the molecular Hamiltonian for N_2.
   - Explain the chemistry-inspired and hardware-friendly local unitary cluster Jastrow (LUCJ) [1]
2. Step 2: Optimize for target hardware
   - Optimize gate counts and layout of the ansatz for hardware execution
3. Step 3: Execute on target hardware
   - Run the optimized circuit on a real QPU to generate samples of the subspace.
4. Step 4: Post-process results
   - Introduce the self-consistent configuration recovery loop [2]
     - Post-process the full set of bitstring samples, using prior knowledge of particle number and the average orbital occupancy calculated on the most recent iteration.
     - Probabilistically create batches of subsamples from recovered bitstrings.
     - Project and diagonalize the molecular Hamiltonian over each sampled subspace.
     - Save the minimum ground state energy found across all batches and update the avg orbital occupancy

We will use several software packages throughout the lesson.

- `PySCF` to define the molecule and setup the Hamiltonian.
- `ffsim` package to construct the LUCJ ansatz.
- `Qiskit` for transpiling the ansatz for hardware execution.

- `Qiskit IBM Runtime` to execute the circuit on a QPU and collect samples.
- `Qiskit Addon SQD` configuration recovery and ground state energy estimation using subspace projection and matrix diagonalization.

1. Map problem to quantum circuits and operators

Molecular Hamiltonian

A molecular Hamiltonian takes the generic form:

$$\hat{H} = \sum_{\substack{pr \\ \sigma}} h_{pr}\, \hat{a}^\dagger_{p\sigma} \hat{a}_{r\sigma} + \sum_{\substack{prqs \\ \sigma\tau}} \frac{(pr|qs)}{2}\, \hat{a}^\dagger_{p\sigma} \hat{a}^\dagger_{q\tau} \hat{a}_{s\tau} \hat{a}_{r\sigma}$$

$\hat{a}^\dagger_{p\sigma}/\hat{a}_{p\sigma}$ are the fermionic creation/annihilation operators associated to the p-th basis set element and the spin σ. h_{pr} and $(pr|qs)$ are the one- and two-body electronic integrals. Using pySCF, we will define the molecule and compute the one- and two-body integrals of the Hamiltonian for basis set `6-31g` .

```python
import warnings

warnings.filterwarnings("ignore")

import pyscf
import pyscf.cc
import pyscf.mcscf

# Specify molecule properties
open_shell = False
spin_sq = 0

# Build N2 molecule
mol = pyscf.gto.Mole()
mol.build(
    atom=[["N", (0, 0, 0)], ["N", (1.0, 0, 0)]], # Tv
    basis="6-31g",
    symmetry="Dooh",
)

# Define active space
```

```
n_frozen = 2
active_space = range(n_frozen, mol.nao_nr())

# Get molecular integrals
scf = pyscf.scf.RHF(mol).run()
num_orbitals = len(active_space)
n_electrons = int(sum(scf.mo_occ[active_space]))
num_elec_a = (n_electrons + mol.spin) // 2
num_elec_b = (n_electrons - mol.spin) // 2
cas = pyscf.mcscf.CASCI(scf, num_orbitals, (num_elec_
mo = cas.sort_mo(active_space, base=0)
hcore, nuclear_repulsion_energy = cas.get_h1cas(mo) #
eri = pyscf.ao2mo.restore(1, cas.get_h2cas(mo), num_o

# Compute exact energy for comparison
```

Output:

```
converged SCF energy = -108.835236570774
CASCI E = -109.046671778080  E(CI) = -32.8155692383188  S^2
```

In this lesson, we will use Jordan-Wigner (JW) transformation to map a fermionic wavefunction to a qubit wavefunction so that it can be prepared using a quantum circuit. The JW transformation maps the Fock space of fermions in M spatial orbitals onto the Hilbert space of 2M qubits, i.e., a spatial orbital is split into two *spin orbitals*, one associated with a spin up (α) electron and another with spin down (β). A spin orbital can be occupied or unoccupied. Usually, when we refer to number of orbitals, we will be using number of *spatial* orbitals. The number of spin orbitals will be double. In quantum circuits, we will represent each spin orbital with one qubit. Thus, a set of qubits will represent spin-up or α-orbitals, and another set will represent spin-up or β-orbitals. For example, N_2 molecule for `6-31g` basis set has 16 spatial orbitals (i.e., $16\ \alpha + 16\ \beta = 32$ spin orbitals). Thus, we will need a 32-qubit quantum circuit (we may need extra ancilla qubits as discussed later). The qubits are measured in computational basis to generate bitstrings, which represent electronic configurations or (Slater) determinants. Throughout this lesson, we will use the terms bitstrings, configurations, and determinants interchangeably. The bitstrings tell us electron occupancy in spin orbitals: a 1 in a bit position means the corresponding spin orbital is occupied, while a 0 means the spin orbital is empty. As electronic structure problems are particle preserving, only a fixed number of spin orbitals must be occupied. The N_2 molecule has 5 spin-up (α) and 5 spin-down (

β) electrons. Thus, any bitstring representing the α and β orbitals must have five $1s$ each for N_2 molecule.

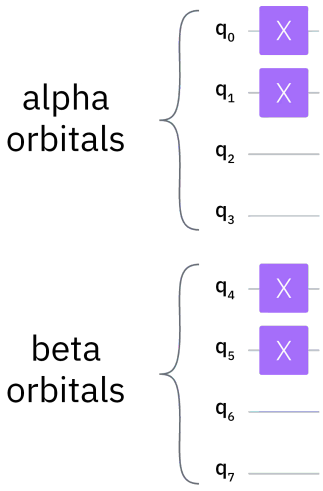
1.1 Quantum circuit for sample generation: The LUCJ ansatz

In this lesson, we will use the local unitary coupled cluster Jastrow (LUCJ) [1] ansatz for quantum state preparation and subsequent sampling. First, we will explain different building blocks of the full UCJ ansatz and the approximations made in the local version of it. Next, by using ffim package, we will construct the LUCJ ansatz and optimize it using Qiskit transpiler for hardware execution.

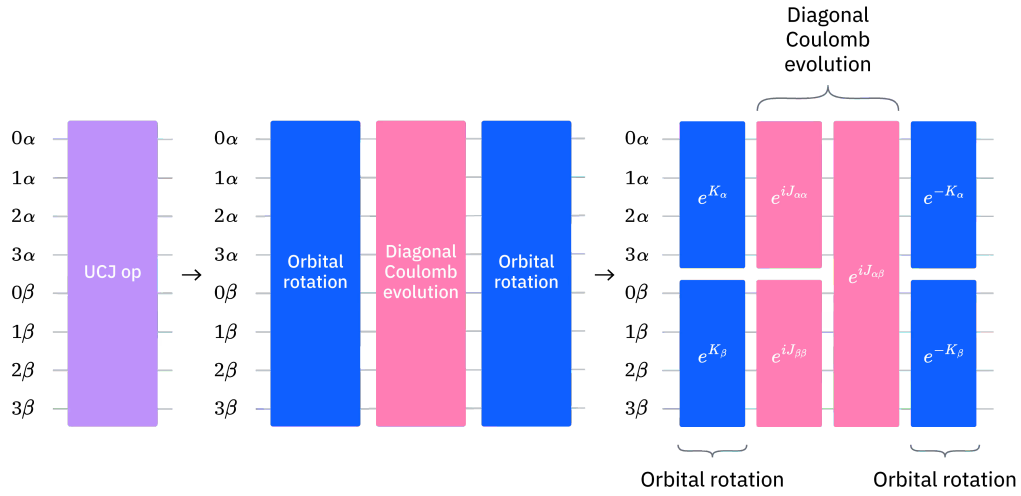
The UCJ ansatz has the following form (for a product of L layers or repetitions of the UCJ operator.)

$$|\psi\rangle = \prod_{\mu=1}^{L} (e^{K^\mu} \times e^{iJ^\mu} \times e^{-K^\mu})|\Phi_0\rangle$$

where, $|\Phi_0\rangle$ is a reference state, typically taken as the Hartree-Fock (HF) state. As the Hartree-Fock state is defined as having the lowest numbered orbitals occupied, the HF state preparation will involve applying X gates to set qubits corresponding to occupied orbitals to one. For example, the HF state preparation block for 4 spatial orbitals and 2 up- and 2 down-spin may look like the following:

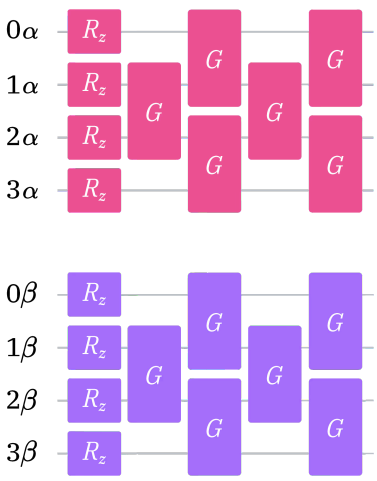


A single repetition of the UCJ operator $\left(e^{K^{(\mu)}} \times e^{iJ^{(\mu)}} \times e^{-K^{(\mu)}}\right)$ consists of a diagonal Coulomb evolution ($e^{iJ^{(\mu)}}$) sandwiched by orbital rotations ($e^{K^{(\mu)}}$ and $e^{-K^{(\mu)}}$).



Orbital rotation blocks work on a single spin species (α (up-spin)/β (down-spin)). For each electron species, orbital rotation consists of a layer of single-qubit R_z gates followed by a sequence of 2-qubit Given's rotation gates ($XX + YY$ gates).

The 2-qubit gates act on adjacent spin-orbitals (nearest neighbor qubits), and therefore, are implementable on IBM QPUs without the need for SWAP gates.

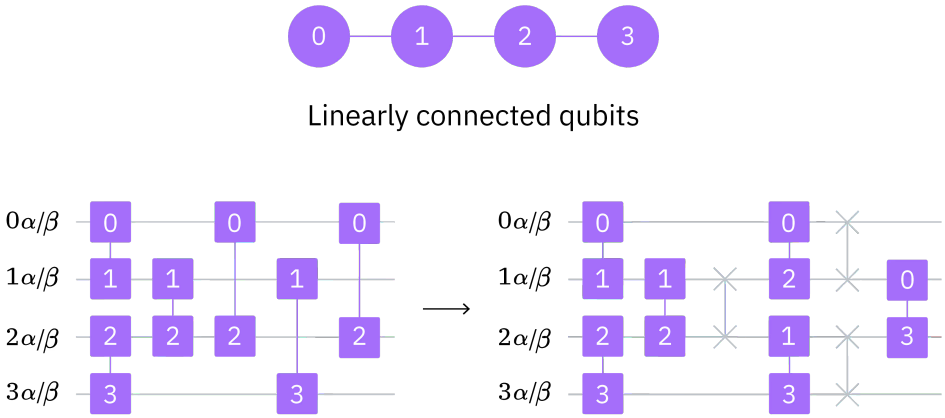


The $e^{iJ^{(\mu)}}$, also known as the diagonal Coulomb operator, consists of three blocks. Two of them work on same spin sectors ($e^{iJ^{(\mu)}_{\alpha\alpha}}$ and $e^{iJ^{(\mu)}_{\beta\beta}}$), and one works between two spin sectors ($e^{iJ^{(\mu)}_{\alpha\beta}}$).

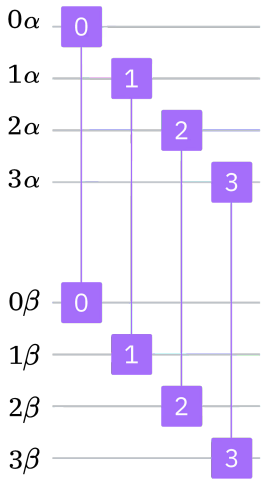
All the blocks in $e^{iJ^{(\mu)}}$ consists of number-number gates $U_{nn}(\phi)$ [1]. A $U_{nn}(\phi)$ gate can be further broken down into a $R_{ZZ}(\frac{\phi}{2})$ gate followed by two single-qubit $Rz(-\frac{\phi}{2})$ gates acting on two separate qubits.

Same-spin components ($J_{\alpha\alpha}$ and $J_{\beta\beta}$) have U_{nn} gates between all possible pairs of qubits. However, as superconducting QPUs have restrictive connectivity, qubits must be swapped to realize gates between non-adjacent qubits.

For example, consider the following $e^{iJ^{(\mu)}_{\alpha\alpha}}$ (or $e^{iJ^{(\mu)}_{\beta\beta}}$) block for $N = 4$ spatial orbitals. For a linear qubit connectivity, the last three gates are not directly implementable as they work between non-adjacent qubits (e.g., Q0 and Q2 are not directly connected). Therefore, we need SWAP gates to make them adjacent (following figure shows an example with 3 SWAP gates).

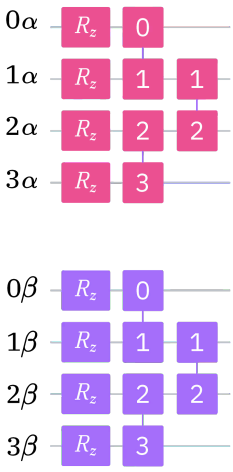


Next, the $J_{\alpha\beta}$ implements gates between same indexed orbitals from different spin sectors (e.g., between 0α and 0β). Similarly, if the qubits are not physically adjacent on a QPU, these gates will also require SWAPs.



From the above discussion, the UCJ ansatz faces some hurdles for HW execution as it needs SWAP gates due to non-adjacent qubit interactions. The local variant of the UCJ ansatz, LUCJ, addresses this challenge by removing some U_{nn} from the diagonal Coulomb operator.

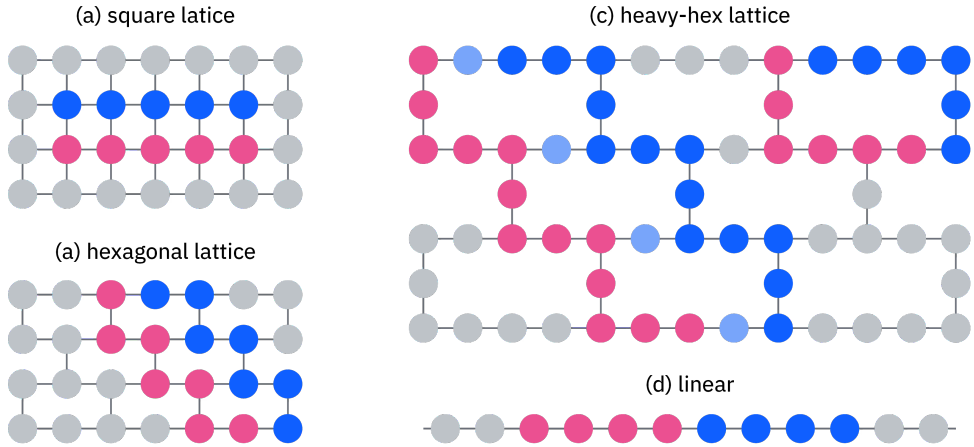
In the same electron species blocks, $J_{\alpha\alpha}$ and $J_{\beta\beta}$), we only keep the U_{nn} gates compatible with nearest-neighbor connectivity and remove gates between non-adjacent qubits in the LUCJ version. Following figure shows the LUCJ block after removal of non-adjacent gates.



Next, the LUCJ version of the $J_{\alpha\beta}$ block that works between different electron species can take different shape based on the device topology.

Here, also, the LUCJ version gets rid of non-compatible gates. The figure below shows variants of the $J_{\alpha\beta}$ block for different qubit topology including grid, hexagonal, heavy-hex, and linear.

- **Grid**: we can have U_{nn} gates between all α and β orbitals without any SWAPs, and therefore, do not need to remove any U_{nn} gates.
- **Hexagonal**: Every other orbital (0th, 2nd, 4th, etc. indexed orbitals) becomes nearest neighbors when α and β are laid out in two adjacent linear chains.
- **Linear**: Only one α and one β orbital are connected, which means the $J_{\alpha\beta}$ block will have only one gate.
- **Heavy-hex**: The α-β interactions are kept between every 4-th indexed (0th, 4th, 8th, etc.) spin orbitals and are need *ancilla* mediated, i.e., we need ancilla qubits between the linear chains representing α and β orbitals. This arrangement needs a limited number of SWAPs.



While removing gates from the UCJ ansatz to construct the LUCJ version makes it more HW compatible, the ansatz loses some expressivity. Therefore, more repetitions (L) of the modified UCJ operator may be needed when using the LUCJ ansatz.

1.2 LUCJ ansatz initialization

The LUCJ is a parameterized ansatz, and we need to initialize the parameters before hardware execution. One way to initialize ansatz is by using `t1` and `t2` amplitudes from classical coupled cluster singles

and doubles (CCSD) method, where `t1` amplitudes are the coefficient of single excitation operators and `t2` amplitudes are for double excitation operators.

Note that while initializing the LUCJ ansatz with `t1` and `t2` amplitudes generate decent results, the ansatz parameters may need further optimization.

```
# Get CCSD t2 amplitudes for initializing the ans
ccsd = pyscf.cc.CCSD(scf, frozen=[i for i in range
ccsd.run()

t1 = ccsd.t1
t2 = ccsd.t2
```

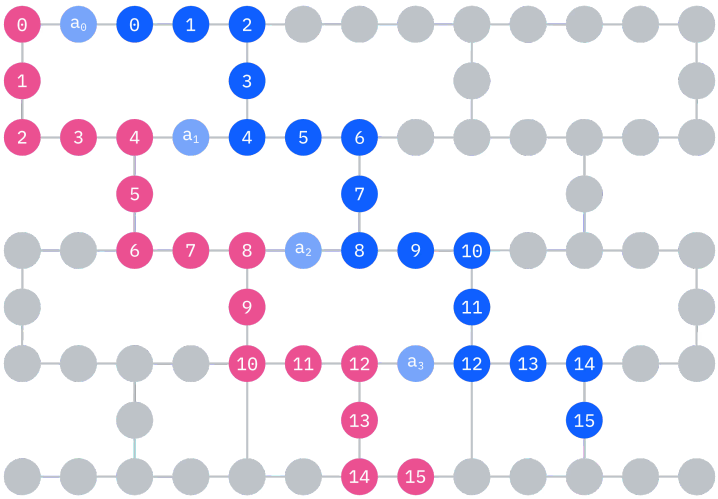
Output:

```
E(CCSD) = -109.0398256929733  E_corr = -0.20458912219883
```

1.3 Constructing the LUCJ ansatz using `ffsim`

We will use the ffsim package to create and initialize the ansatz with `t1` and `t2` amplitudes computed above. Since our molecule has a closed-shell Hartree-Fock state, we will use the spin-balanced variant of the UCJ ansatz, UCJOpSpinBalanced.

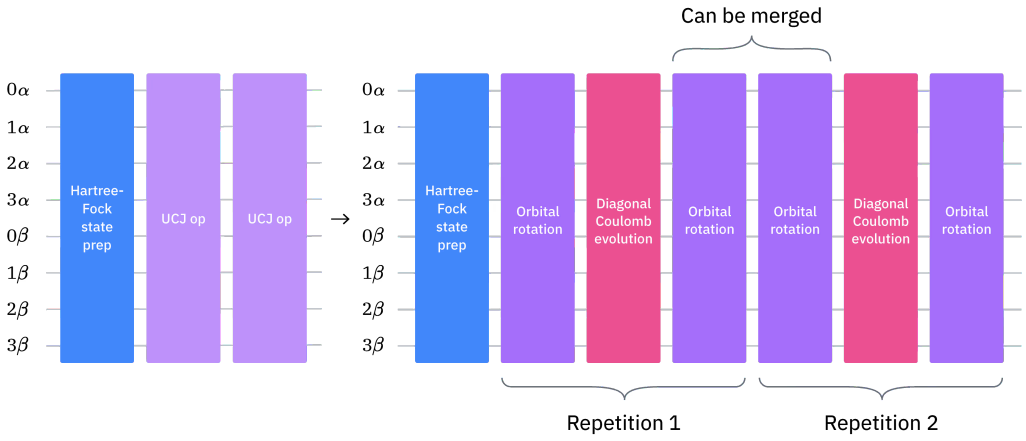
As IBM hardware has a heavy-hex topology, we will adopt the *zig-zag* pattern used in [1] and explained above for qubit interactions. In this pattern, orbitals (qubits) with the same spin are connected with a line topology (red and blue circles). Due to the heavy-hex topology, orbitals for different spins have connections between every 4th orbital (0th, 4th, 8th, etc.) (purple circles).




```python
import ffsim
from qiskit import QuantumCircuit, QuantumRegister

n_reps = 2
alpha_alpha_indices = [(p, p + 1) for p in range(num_
alpha_beta_indices = [(p, p) for p in range(0, num_or

ucj_op = ffsim.UCJOpSpinBalanced.from_t_amplitudes(
    t2=t2,
    t1=t1,
    n_reps=n_reps,
    interaction_pairs=(alpha_alpha_indices, alpha_bet
)

nelec = (num_elec_a, num_elec_b)

# create an empty quantum circuit
qubits = QuantumRegister(2 * num_orbitals, name="q")
circuit = QuantumCircuit(qubits)

# prepare Hartree-Fock state as the reference state a
circuit.append(ffsim.qiskit.PrepareHartreeFockJW(num_

# apply the UCJ operator to the reference state
circuit.append(ffsim.qiskit.UCJOpSpinBalancedJW(ucj_o
circuit.measure_all()
# circuit.decompose().draw("mpl", scale=0.5, fold=-1)
```

No output produced

The LUCJ ansatz with repeated layers can be optimized by merging some adjacent blocks. Consider a case for `n_reps=2`. The two orbital rotation blocks in the middle can be merged into a single orbital rotation block. The `ffsim` package has a pass manager named `ffsim.qiskit.PRE_INIT` to optimize the circuit by merging such adjacent blocks.



2. Optimize for target hardware

First, we fetch a backend of our choice. We will optimize our circuit for the backend, and then execute the optimized circuit on the same backend to generate samples for the subspace.

```
from qiskit_ibm_runtime import QiskitRuntimeServi
from qiskit_ibm_runtime import QiskitRuntimeService

service = QiskitRuntimeService()
backend = service.backend("ibm_kyiv")
```

No output produced

Next, we recommend the following steps to optimize the ansatz and make it hardware-compatible.

– Select physical qubits (`initial_layout`) from the target hardware that adheres to the zig-zag pattern (two linear chains with ancilla qubit in-between them) described above. Laying out qubits in this pattern leads to an efficient hardware-compatible circuit with less gates.

– Generate a staged pass manager using the generate_preset_pass_manager function from Qiskit with your choice of `backend` and `initial_layout` .

– Set the `pre_init` stage of your staged pass manager to `ffsim.qiskit.PRE_INIT` . `ffsim.qiskit.PRE_INIT` includes Qiskit transpiler passes that decompose gates into orbital rotations and then merges the orbital rotations, resulting in fewer gates in the final circuit.

– Run the pass manager on your circuit.

```
from qiskit.transpiler.preset_passmanagers import

spin_a_layout = [0, 14, 18, 19, 20, 33, 39, 40, 41, 5
spin_b_layout = [2, 3, 4, 15, 22, 23, 24, 34, 43, 44,

initial_layout = spin_a_layout + spin_b_layout

pass_manager = generate_preset_pass_manager(
    optimization_level=3, backend=backend, initial_la
)

# without PRE_INIT passes
isa_circuit = pass_manager.run(circuit)
print(f"Gate counts (w/o pre-init passes): {isa_circu

# with PRE_INIT passes
# We will use the circuit generated by this pass mana
pass_manager.pre_init = ffsim.qiskit.PRE_INIT
isa_circuit = pass_manager.run(circuit)
print(f"Gate counts (w/ pre-init passes): {isa_circui
```

Output:

```
Gate counts (w/o pre-init passes): OrderedDict({'rz': 7579,
Gate counts (w/ pre-init passes): OrderedDict({'rz': 4088,
```

3. Execute on target hardware

After optimizing the circuit for hardware execution, we are ready to run it on the target hardware and collect samples for ground state energy estimation. As we only have one circuit, we will use Qiskit Runtime's Job execution mode and execute our circuit.

```
from qiskit_ibm_runtime import SamplerV2 as Sampl

sampler = Sampler(mode=backend)
sampler.options.dynamical_decoupling.enable = True

job = sampler.run([isa_circuit], shots=10_000) # Takes
```

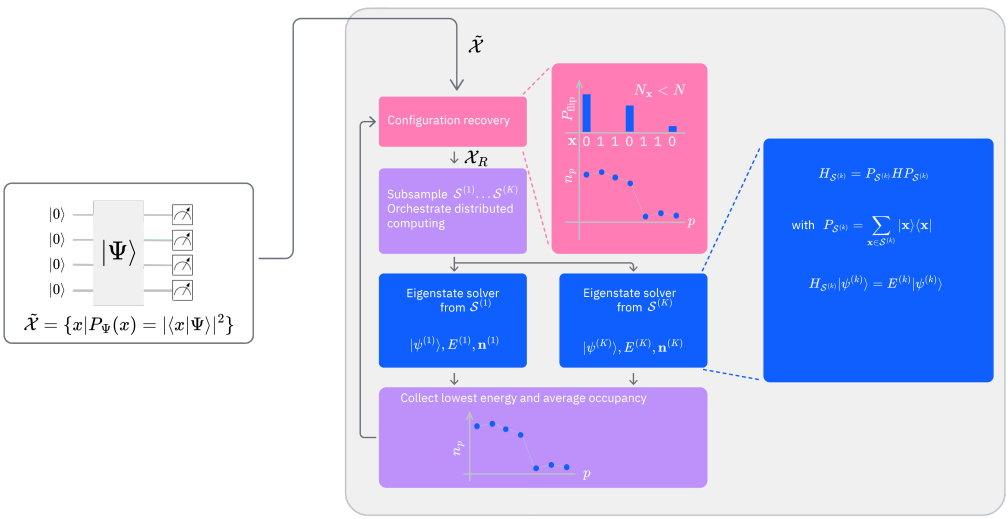
No output produced

```
# Run cell after IQX job completion
primitive_result = job.result()
pub_result = primitive_result[0]
counts = pub_result.data.meas.get_counts()
```

No output produced

4. Post-process results

The post-processing part of the SQD workflow can be summarized using the following diagram.



Sampling the LUCJ ansatz in the computational basis generates a pool of noisy configurations $\tilde{\chi}$, which are used in the post-processing routine. It involves a method called (details discussed later) *configuration recovery* to probabilistically correct configurations with incorrect electron numbers. Configurations only with correct electron numbers $\tilde{\chi}_R$ are then subsampled and distributed into multiple batches based on the frequency of appearance of each unique configuration. Each batch of samples defines a subspace ($\mathcal{S}^{(k)}$). Next, the molecular Hamiltonian, H, is projected onto subspaces:

$$H_{\mathcal{S}^{(k)}} = P_{\mathcal{S}^{(k)}} H_{\mathcal{S}^{(k)}} \text{ with } P_{\mathcal{S}^{(k)}} = \sum_{x \in \mathcal{S}^{(k)}} |x\rangle\langle x|$$

Each projected Hamiltonian $H_{\mathcal{S}^{(k)}}$ is then fed into an Eigensolver, where it is diagonalized to compute eigenvalues and eigenvectors to reconstruct an eigenstate. In this lesson, we project and diagonalize the Hamiltonian using the `qiskit-addon-sqd` package which uses the Davidson's method from PySCF for diagonalization.

$$H_{\mathcal{S}^{(k)}} |\psi^{(k)}\rangle = E^{(k)} |\psi^{(k)}\rangle$$

We then collect the lowest eigenvalue (energy) from the batches, and also compute average orbital occupancy, $\mathbf{n}$. The average occupancy information is used in the configuration recovery step to probabilistically correct noise configurations.

Next, we explain the self-consistent configuration recovery loop in detail and show concrete code examples to implement the above-mentioned steps to estimate the ground state energy of N_2 Hamiltonian.

4.1 Configuration recovery: overview

Each bit in a bitstring (Slater determinant) represents a spin orbital. The right half of a bitstring represents spin-up orbitals, and the left half represents spin-down orbitals. A `1` means the orbital is occupied by an electron, and a `0` means the orbital is empty. We know the correct number of particles (both up-spin electron and down-spin electron) a priori. Suppose we have a determinant x with N_x electrons (i.e., there are N_x numbers of 1s in the bitstring) in it. The correct number of particles is N. If $N_x \neq N$, then we know that the bitstring is corrupted by noise. The self-consistent configuration routine attempts to correct the bitstring by probabilistically flipping $|N_x - N|$ bits by leveraging average orbital occupancy information. The average orbital occupancy (n

) tells us how likely an orbital be occupied by an electron. If $N_x < N$, we have fewer electrons and need to flip some 0s to 1s and vice versa.

The probability of flipping can be $|x[i] - avg\_occupancy[i]|$ for `i` -th spin orbital. In [2], the authors used a weighted probability of flipping using modified ReLU function.

$$w(y) = \begin{cases} \delta \frac{y}{h} & \text{if } y \leq h \\ \delta + (1-\delta)\frac{y-h}{1-h} & \text{if } y > h \end{cases}$$

Here h defines the location of "corner" of the ReLU function, and the parameter δ defines the value of the ReLU function at the corner. For $\delta = 0$, w becomes true ReLU function, and for $\delta > 0$, it becomes *modified* ReLU. In the paper, the authors used $\delta = 0.01$ and $h =$ number of alpha (or beta) particles/number of alpha (or beta) spin orbitals $= N/M$ (filling factor).

The average orbital occupancy (n) is not known a priori. The first iteration of the ground state estimation starts with configurations with only correct particle numbers in both spin species. After the first iteration, we have an estimate of the ground state, and using the estimate, we can construct the first guess of n. This guess of n is used to recover configurations, run the next iteration of ground state estimation, and self-consistently refine the guess of n. The process repeats until the a stopping criterion is met.

Consider the following example for $N = 2$ and $x = |1000\rangle$ ($N_x = 1$). We need to flip one of the 0s to 1 to correct it for particle numbers, and the choices are `1100` , `1010` , and `1001` . Based on the probability of flipping, one of the choices will be selected as *recovered configuration* (or the bitstring with correct number of particles).

Suppose in the first iteration we run two batches, and the estimated ground states from them are:

$$\text{Batch0: } |\psi\rangle = 0.8 \times |1001\rangle + 0.6 \times |0101\rangle$$
$$\text{Batch1: } |\psi\rangle = \frac{1}{\sqrt{3}}\left(|1001\rangle + |0101\rangle + |0110\rangle\right)$$

Using the computational basis states and their amplitudes, we can compute probability of electron occupancies (in short *occupancies*) per spin-orbital (qubit) (note that probability = |amplitude|^2). Below we tabulate qubit-wise occupancies for each bitstring appearing in the estimated ground state and compute total orbital occupancy for a batch. Note that, as per Qiskit ordering convention, the right most bit represents qubit-0 (Q0), and the left most bit represents Q3.

Occupancy (Batch0):

	Q3	Q2	Q1	Q0
1001	0.64	0.0	0.0	0.64
0110	0.0	0.36	0.36	0.0
n *(Batch0)*	**0.64**	**0.36**	**0.36**	**0.64**

Occupancy (Batch1)

	Q3	Q2	Q1	Q0
1001	0.33	0.00	0.00	0.33
0101	0.0	0.33	0.00	0.33
0110	0.0	0.33	0.33	0.00
n *(Batch1)*	**0.33**	**0.66**	**0.33**	**0.66**

Occupancy (average across batches)

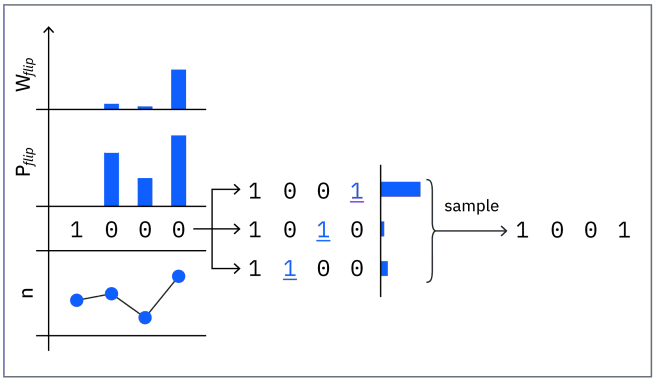
	Q3	Q2	Q1	Q0
n *(Batch0)*	0.64	0.36	0.36	0.64
n *(Batch1)*	0.33	0.66	0.33	0.66
n *(average)*	**0.49**	**0.51**	**0.35**	**0.65**

Using the average orbital occupancy computed above, we can find the probabilities of flip for different orbitals in the configuration $x = |1000\rangle$. As the orbital represented by Q3 is already occupied and need not to be flipped, we set its p(flip) to 0. For the remaining orbitals, which are unoccupied, the probability of flip is $|x[i] - \mathbf{n}[i]|$ each. Along with p(flip), we also compute the probability weight associated with flipping using the modified ReLU function described above.

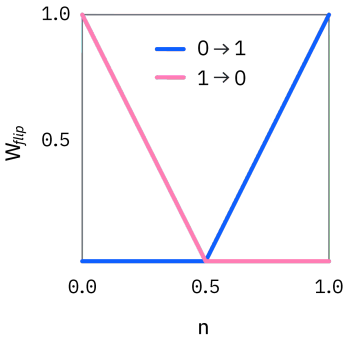
Probability of flip ($x = |1000\rangle, \delta = 0.01, h = N/M = 2/4 = 0.50$)

	Q3	Q2	Q1	Q0
p(flip) ($\lvert x[i] - \mathbf{n}[i]\rvert$)	0	0.51	0.35	0.65
w(p(flip))	0	0.03	0.007	0.31

Finally, using weighted probabilities above, we can flip one of the unoccupied Q2, Q1, and Q0 orbitals. Based on the values above, Q0 will be flipped most likely, and a possible recovered configuration can be $|1001\rangle$.



Configuration recovery



Probability weight associated with flipping

The complete self-consistent configuration recovery process can be summarized as follows:

First iteration: Suppose the bitstrings (configurations or Slater determinants) generated by the quantum computer form a set $\widetilde{\chi}$, which includes both configurations with correct ($\widetilde{\chi}_{correct}$) and incorrect ($\widetilde{\chi}_{incorrect}$) number of particles in each spin sector.

1. Configurations from ($\widetilde{\chi}_{correct}$) are randomly sampled to create batches $\left(\mathcal{S}^{(1)}, \cdots, \mathcal{S}^{(K)}\right)$ of vectors for subspace projection. The number of batches and samples in each batch are user defined parameters. The larger the number of samples in each batch, the larger the subspace dimension and more computationally demanding the diagonalization becomes. On the other hand, too small number of samples may miss the ground state support vectors and lead to incorrect estimation.
2. Run the eigenstate solver (i.e. projection onto subspace and diagonalization) on the batches and obtain approximate eigenstates. $|\psi^{(1)}\rangle, \cdots, |\psi^{(K)}\rangle$.
3. From the approximate eigenstates construct the first guess for n.

Subsequent iterations:

1. Using n correct the configurations with wrong particle number in $\widetilde{\chi}_{incorrect}$. Suppose we name them $\widetilde{\chi}_{correct\_new}$. Then, $\widetilde{\chi}_{recovered}(\widetilde{\chi}_R) = \widetilde{\chi}_{correct} \cup \widetilde{\chi}_{correct\_new}$ forms the new set of configurations with correct particle numbers.
2. $\widetilde{\chi}_R$ is sampled to create batches $\mathcal{S}^{(1)}, \cdots, \mathcal{S}^{(K)}$.

3. Eigenstate solver runs with new batches and generates new estimates of ground states $|\psi^{(1)}\rangle, \cdots, |\psi^{(K)}\rangle$.
4. From the approximate eigenstates construct refined guess for n.
5. If the stopping criterion is not met, go back to step `2.1` .

4.2 Ground state estimation

First, we will transform the counts into a bitstring matrix and probability array for post-processing.

Each row in the matrix represents one unique bitstring. Since qubits are indexed from the right of a bitstring in Qiskit, column `0` represents qubit `N-1` , and column `N-1` represents qubit `0` , where `N` is the number of qubits.

The alpha orbitals are represented in the column index range `(N, N/2]` (right half), and the beta orbitals are represented in the column range `(N/2, 0]` (left half).

```
from qiskit_addon_sqd.counts import counts_to_arr

# Convert counts into bitstring and probability arrays
bitstring_matrix_full, probs_arr_full = counts_to_arra
```

No output produced

There are a few user-controlled options which are important for this technique:

- `iterations` : Number of self-consistent configuration recovery iterations
- `n_batches` : Number of batches of configurations used by the different calls to the eigenstate solver
- `samples_per_batch` : Number of unique configurations to include in each batch
- `max_davidson_cycles` : Maximum number of Davidson cycles run by each eigensolver

```
import numpy as np
from qiskit_addon_sqd.configuration_recovery import
from qiskit_addon_sqd.fermion import (
    bitstring_matrix_to_ci_strs,
```

```
    solve_fermion,
)
from qiskit_addon_sqd.subsampling import postselect_a

rng = np.random.default_rng(24)
# SQD options
iterations = 5

# Eigenstate solver options
n_batches = 5
samples_per_batch = 500
max_davidson_cycles = 300

# Self-consistent configuration recovery loop
e_hist = np.zeros((iterations, n_batches))  # energy
s_hist = np.zeros((iterations, n_batches))  # spin hi
occupancy_hist = []
avg_occupancy = None
for i in range(iterations):
    print(f"Starting configuration recovery iteratior
    # On the first iteration, we have no orbital occu
    # solver, so we begin with the full set of noisy
    if avg_occupancy is None:
        bs_mat_tmp = bitstring_matrix_full
        probs_arr_tmp = probs_arr_full

    # If we have average orbital occupancy informatic
    else:
        bs_mat_tmp, probs_arr_tmp = recover_configura
            bitstring_matrix_full,
            probs_arr_full,
            avg_occupancy,
            num_elec_a,
            num_elec_b,
            rand_seed=rng,
        )

    # Create batches of subsamples. We post-select he
    # with incorrect hamming weight during iteration
    batches = postselect_and_subsample(
        bs_mat_tmp,
        probs_arr_tmp,
        hamming_right=num_elec_a,
        hamming_left=num_elec_b,
        samples_per_batch=samples_per_batch,
        num_batches=n_batches,
        rand_seed=rng,
    )

    # Run eigenstate solvers in a loop. This loop shc
    e_tmp = np.zeros(n_batches)
    s_tmp = np.zeros(n_batches)
```

```
        occs_tmp = []
        coeffs = []
        for j in range(n_batches):
            strs_a, strs_b = bitstring_matrix_to_ci_strs(
            print(f"  Batch {j} subspace dimension: {len(
            energy_sci, coeffs_sci, avg_occs, spin = solv
                batches[j],
                hcore,
                eri,
                open_shell=open_shell,
                spin_sq=spin_sq,
                max_davidson=max_davidson_cycles,
            )
            energy_sci += nuclear_repulsion_energy
            e_tmp[j] = energy_sci
            s_tmp[j] = spin
            occs_tmp.append(avg_occs)
            coeffs.append(coeffs_sci)

        # Combine batch results
        avg_occupancy = tuple(np.mean(occs_tmp, axis=0))

        # Track optimization history
        e_hist[i, :] = e_tmp
        s_hist[i, :] = s_tmp
```

Output:

```
Starting configuration recovery iteration 0
  Batch 0 subspace dimension: 21609
  Batch 1 subspace dimension: 21609
  Batch 2 subspace dimension: 21609
  Batch 3 subspace dimension: 21609
  Batch 4 subspace dimension: 21609
Starting configuration recovery iteration 1
  Batch 0 subspace dimension: 609961
  Batch 1 subspace dimension: 616225
  Batch 2 subspace dimension: 627264
  Batch 3 subspace dimension: 633616
  Batch 4 subspace dimension: 624100
Starting configuration recovery iteration 2
  Batch 0 subspace dimension: 564001
  Batch 1 subspace dimension: 605284
  Batch 2 subspace dimension: 582169
  Batch 3 subspace dimension: 559504
  Batch 4 subspace dimension: 591361
Starting configuration recovery iteration 3
  Batch 0 subspace dimension: 550564
  Batch 1 subspace dimension: 549081
  Batch 2 subspace dimension: 531441
  Batch 3 subspace dimension: 527076
```

```
  Batch 4 subspace dimension: 531441
Starting configuration recovery iteration 4
  Batch 0 subspace dimension: 544644
  Batch 1 subspace dimension: 580644
  Batch 2 subspace dimension: 527076
  Batch 3 subspace dimension: 531441
  Batch 4 subspace dimension: 537289
```

4.3 Discussion of results

The first plot shows that after a few iterations we estimate the ground state energy within ~24 mH (chemical accuracy is typically accepted to be 1 kcal/mol $\approx$ 1.6 mH). The second plot shows the average occupancy of each spatial orbital after the final iteration. We can see that both the spin-up and spin-down electrons occupy the first five orbitals with high probability in our solutions.

Although the estimated ground state energy is decent, it is not within the chemical accuracy limit ($\pm \approx 1.6$ mH). This gap can be attributed to the small subspace dimension we used above for projection and diagonalization. As we used `samples_per_batch=500`, the subspace is spanned by max 500 vectors, which is missing vectors from ground state support. Increasing the `samples_per_batch` parameter should improve the accuracy at the expense of more classical compute resources and runtime.

```
# Data for energies plot
x1 = range(iterations)
min_e = [np.min(e) for e in e_hist]
e_diff = [abs(e - exact_energy) for e in min_e]
yt1 = [1.0, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5]

# Chemical accuracy (+/- 1 milli-Hartree)
chem_accuracy = 0.001

# Data for avg spatial orbital occupancy
y2 = occupancy_hist[-1][0] + occupancy_hist[-1][1]
x2 = range(len(y2))
```

No output produced

```
import matplotlib.pyplot as plt

```

```
fig, axs = plt.subplots(1, 2, figsize=(12, 6))

# Plot energies
axs[0].plot(x1, e_diff, label="energy error", marker=
axs[0].set_xticks(x1)
axs[0].set_xticklabels(x1)
axs[0].set_yticks(yt1)
axs[0].set_yticklabels(yt1)
axs[0].set_yscale("log")
axs[0].set_ylim(1e-6)
axs[0].axhline(y=chem_accuracy, color="#BF5700", line
axs[0].set_title("Approximated Ground State Energy Er
axs[0].set_xlabel("Iteration Index", fontdict={"fonts
axs[0].set_ylabel("Energy Error (Ha)", fontdict={"for
axs[0].legend()

# Plot orbital occupancy
axs[1].bar(x2, y2, width=0.8)
axs[1].set_xticks(x2)
axs[1].set_xticklabels(x2)
axs[1].set_title("Avg Occupancy per Spatial Orbital")
axs[1].set_xlabel("Orbital Index", fontdict={"fontsiz
axs[1].set_ylabel("Avg Occupancy", fontdict={"fontsiz

print(f"Exact energy: {exact_energy:.5f} Ha")
print(f"SQD energy: {min_e[-1]:.5f} Ha")
print(f"Absolute error: {e_diff[-1]:.5f} Ha")
plt.tight_layout()
plt.show()
```

Output:

```
Exact energy: -109.04667 Ha
SQD energy: -109.02234 Ha
Absolute error: 0.02434 Ha
```