

Molecular Ground State & Excited State Simulation via VQE

Project Vision

Hypothesis: Ground state wavefunctions of molecular systems can be computed using pre-optimized template components, integrated according to molecule type and atomic environment, with final optimization. Excited states can be modeled using hybrid classical-quantum geometric techniques to reproduce spectroscopic transitions.

- Goal:** Implement a Variational Quantum Eigensolver (VQE) pipeline for fermionic systems that:
1. Maps Fermionic Hamiltonians to Qubit Hamiltonians
 2. Uses pre-optimized ansatz templates based on atomic environment
 3. Optimizes variational parameters using deep learning
 4. Extends to excited states for spectroscopic predictions
 5. Maps phase diagrams of molecular Hamiltonians

Location: [qward/examples/papers/molecular-vqe/](#)

Scientific Background

The Challenge

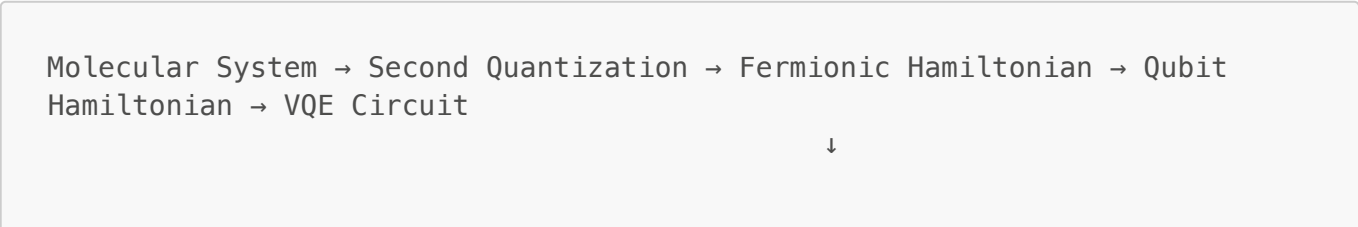
Quantum simulation of molecular systems faces NISQ-era limitations:

- Limited qubit counts restrict molecular orbital basis size
- Gate errors accumulate in deep circuits required for chemical accuracy
- Variational algorithms suffer from barren plateaus

Our Approach

1. **Template-based ansatz:** Pre-optimize circuit components for common atomic environments (H, C, N, O, etc.), reducing variational parameter space
2. **Deep learning optimization:** Replace classical optimizers (COBYLA, SPSA) with neural network-based parameter prediction
3. **Geometric excited states:** Use quantum subspace expansion (QSE) and equation-of-motion (EOM) techniques for excited states
4. **Incremental validation:** Start with H₂, HeH⁺, LiH before scaling to larger molecules

Key Transformations



Jordan-Wigner / Bravyi-Kitaev / Parity

mapping

Team & Responsibilities

Agent	Phase(s)	Primary Deliverables
quantum-research-lead	1, 6	Strategic direction, feasibility, synthesis
quantum-computing-researcher	2	Hamiltonian formulation, ansatz design, mapping proofs
test-engineer	3	TDD test suite for all components
python-architect	4	VQE implementation, deep learning integration
quantum-data-scientist	5	Analysis, phase diagrams, spectroscopic validation
All	6	Review, iteration, final validation

Phase 1: Ideation & Feasibility (Lead + Researcher)

Objective

Scope the project, assess NISQ feasibility, define validation milestones.

Tasks

Research Lead:

- ☐ Literature review: Current state of molecular VQE (2024-2025 papers)
- ☐ Assess quantum advantage potential for target molecules
- ☐ Define validation ladder: $H_2 \rightarrow HeH^+ \rightarrow LiH \rightarrow H_2O \rightarrow$ larger systems
- ☐ Identify NISQ constraints for each target molecule
- ☐ Establish success criteria tied to chemical accuracy (1.6 mHartree)

Researcher (support):

- ☐ Qubit requirements per molecule (minimal basis vs STO-3G vs 6-31G)
- ☐ Circuit depth estimates for UCCSD vs hardware-efficient ansätze
- ☐ Barren plateau risk assessment for deep learning optimization
- ☐ Survey fermion-to-qubit mappings (JW, BK, parity) trade-offs

Deliverables

molecular-vqe/
phase1_feasibility_study.md # Feasibility assessment
phase1_literature_review.md # Key papers and findings
phase1_validation_ladder.md # Incremental validation plan

Validation Ladder (Start Small)

Level	System	Qubits (STO-3G)	Electrons	Classical Reference
1	H ₂	4	2	-1.1373 Ha (FCI)
2	HeH ⁺	4	2	-2.8627 Ha (FCI)
3	LiH	12	4	-7.8825 Ha (FCI)
4	H ₂ O	14	10	-75.0115 Ha (CCSD(T))
5	NH ₃	16	10	TBD

Success Criteria

Metric	Threshold	Notes
Ground state energy error	≤ 1.6 mHa (chemical accuracy)	vs FCI/CCSD(T)
Excited state error	≤ 0.1 eV	For spectroscopic transitions
Circuit depth	≤ 100 (NISQ-compatible)	After transpilation
Deep learning convergence	≤ 50 epochs	vs 200+ optimizer iterations

Risk Assessment

Risk	Impact	Mitigation
Barren plateaus in deep ansatz	High	Use hardware-efficient ansatz, layer-wise training
Qubit count limits molecular size	Medium	Use active space reduction, frozen core approximation
Gate noise destroys chemical accuracy	High	Error mitigation (ZNE, PEC), noise-aware training
Deep learning overfits to molecules	Medium	Pre-train on diverse molecular fragments

Handoff Checklist

- ☐ Feasibility study approved
- ☐ Validation ladder defined with classical references
- ☐ NISQ constraints documented
- ☐ Risk mitigation strategies identified

Phase 2: Theoretical Design (Researcher)

Objective

Design the complete VQE pipeline: Hamiltonian construction, ansatz architecture, and optimization strategy.

Tasks

2.1 Molecular Hamiltonian Construction

- ☐ Define second quantization formalism
- ☐ Implement molecular orbital integrals (one-electron h_{pq} , two-electron h_{pqrs})
- ☐ Construct fermionic Hamiltonian in creation/annihilation operators

2.2 Fermion-to-Qubit Mapping

- ☐ Implement Jordan-Wigner transformation
- ☐ Implement Bravyi-Kitaev transformation
- ☐ Implement Parity mapping with two-qubit reduction
- ☐ Prove equivalence of eigenspectra under mappings

2.3 Ansatz Design

- ☐ Define UCCSD (Unitary Coupled Cluster) ansatz
- ☐ Define hardware-efficient ansatz (HEA) alternative
- ☐ Design template-based ansatz for atomic environments
- ☐ Analyze expressibility vs trainability trade-off

2.4 Deep Learning Optimization

- ☐ Design neural network architecture for parameter prediction
- ☐ Define loss function incorporating energy variance
- ☐ Design pre-training strategy on molecular fragments
- ☐ Integrate with VQE measurement feedback

2.5 Excited State Methods

- ☐ Formulate Quantum Subspace Expansion (QSE)
- ☐ Formulate Equation-of-Motion (EOM-VQE)
- ☐ Design measurement strategy for transition amplitudes

Deliverables

```
molecular-vqe/  
  phase2_theoretical_design.md      # Complete theoretical specification  
  phase2_hamiltonian_formulation.md # Detailed Hamiltonian construction  
  phase2_ansatz_design.md           # Ansatz architecture details  
  phase2_deep_learning_spec.md      # Neural network specification
```

Mathematical Specifications

Molecular Hamiltonian (Second Quantization)

$$H = \sum_{pq} h_{pq} a_{\dagger p} a_q + (1/2) \sum_{pqrs} h_{pqrs} a_{\dagger p} a_{\dagger q} a_r a_s + E_{\text{nuc}}$$

where:

- $h_{pq} = \langle p|h|q \rangle$ (one-electron integrals)
- $h_{pqrs} = \langle pq|rs \rangle$ (two-electron integrals)
- E_{nuc} = nuclear repulsion energy

Jordan-Wigner Transformation

$$a_{\dagger j} = (1/2)(X_j - iY_j) \otimes Z_{\{j-1\}} \otimes \dots \otimes Z_0$$

$$a_j = (1/2)(X_j + iY_j) \otimes Z_{\{j-1\}} \otimes \dots \otimes Z_0$$

$$\text{Number operator: } n_j = (1/2)(I - Z_j)$$

UCCSD Ansatz

$$|\psi(\theta)\rangle = e^{\{T(\theta) - T_{\dagger}(\theta)\}} |HF\rangle$$

$$T(\theta) = \sum_{ia} \theta^a a_{\dagger i} a_a + \sum_{ijab} \theta^{ab}_{ij} a_{\dagger i} a_{\dagger j} a_b a_a$$

where:

- $|HF\rangle$ = Hartree-Fock reference state
- i, j = occupied orbitals
- a, b = virtual orbitals

Template-Based Ansatz (Novel Contribution)

$$U_{\text{template}}(\theta) = \prod_{\text{env}} U_{\text{env}}(\theta_{\text{env}})$$

where U_{env} are pre-optimized unitary blocks for:

- U_{H} : Hydrogen environment
- $U_{\text{C}_{\text{sp}^3}}$: Carbon sp^3 hybridization
- $U_{\text{C}_{\text{sp}^2}}$: Carbon sp^2 hybridization
- U_{N} : Nitrogen environment
- U_{O} : Oxygen environment

Deep Learning Parameter Prediction

$$\theta_{\text{predicted}} = \text{NN}(\text{molecular_features})$$

Input features:

- Atom types and positions

- Bond connectivity (adjacency matrix)
- Initial Hartree-Fock parameters
- Molecular fingerprints (e.g., Coulomb matrix)

Architecture:

- Graph Neural Network (GNN) for molecular structure
- Attention mechanism for orbital interactions
- Output: θ parameters for ansatz

Quantum Subspace Expansion (Excited States)

$H_{\text{sub}} = \langle \psi_i | H | \psi_j \rangle$ where $|\psi_j\rangle = 0_j |\psi_0\rangle$

Operators 0_j : $\{I, a^\dagger_a a_i, a^\dagger_a a^\dagger_b a_j a_i, \dots\}$

Excited state energies: Eigenvalues of H_{sub}

Handoff Checklist

- ☐ Hamiltonian construction formulas complete
- ☐ All fermion-to-qubit mappings specified with proofs
- ☐ Ansatz circuits diagrammed
- ☐ Deep learning architecture specified
- ☐ Excited state method formulated
- ☐ Expected energy ranges for all validation molecules

Phase 3: Test Design (Test Engineer)

Objective

Create comprehensive TDD test suite that defines expected behavior for all components.

Tasks

- ☐ Create test fixtures for validation molecules (H_2 , HeH^+ , LiH)
- ☐ Write Hamiltonian construction tests (validated against PySCF/OpenFermion)
- ☐ Write fermion-to-qubit mapping tests
- ☐ Write ansatz expressibility tests
- ☐ Write VQE convergence tests
- ☐ Write deep learning integration tests
- ☐ Write excited state tests

Deliverables

```
molecular-vqe/  
  tests/
```

conftest.py	# Shared fixtures (molecules, backends)
test_molecular_integrals.py	# One/two-electron integral tests
test_fermionic_hamiltonian.py	# Second quantization tests
test_qubit_mapping.py	# JW, BK, Parity mapping tests
test_ansatz.py	# UCCSD, HEA, template ansatz tests
test_vqe_ground_state.py	# Ground state energy tests
test_vqe_excited_state.py	# QSE/EOM excited state tests
test_deep_learning_optimizer.py	# Neural network optimizer tests
test_classical_baseline.py	# PySCF/FCI validation (must pass)
test_phase_diagram.py	# Hamiltonian phase diagram tests

Test Categories

1. Classical Baseline Tests (Must Pass First)

```
import pytest
import numpy as np
from pyscf import gto, scf, fci

@pytest.fixture
def h2_molecule():
    """H2 molecule at equilibrium bond length."""
    mol = gto.M(
        atom='H 0 0 0; H 0 0 0.74',
        basis='sto-3g',
        unit='angstrom'
    )
    return mol

@pytest.fixture
def h2_fci_energy(h2_molecule):
    """FCI ground state energy for H2."""
    mf = scf.RHF(h2_molecule).run()
    cisolver = fci.FCI(mf)
    e_fci, _ = cisolver.kernel()
    return e_fci # Expected: -1.1373 Ha

def test_h2_fci_reference(h2_fci_energy):
    """Validate FCI reference energy for H2."""
    assert np.isclose(h2_fci_energy, -1.1373, atol=0.001)

@pytest.fixture
def lih_molecule():
    """LiH molecule at equilibrium."""
    mol = gto.M(
        atom='Li 0 0 0; H 0 0 1.6',
        basis='sto-3g',
        unit='angstrom'
    )
    return mol
```

2. Hamiltonian Construction Tests

```
def test_one_electron_integrals_hermitian(h2_molecule):
    """One-electron integrals must be Hermitian."""
    h1 = compute_one_electron_integrals(h2_molecule)
    assert np.allclose(h1, h1.T.conj())

def test_two_electron_integrals_symmetry(h2_molecule):
    """Two-electron integrals have 8-fold symmetry."""
    h2 = compute_two_electron_integrals(h2_molecule)
    # (pq|rs) = (qp|rs) = (pq|sr) = (rs|pq) ...
    assert np.allclose(h2, h2.transpose(1,0,2,3))
    assert np.allclose(h2, h2.transpose(0,1,3,2))
    assert np.allclose(h2, h2.transpose(2,3,0,1))

def test_fermionic_hamiltonian_particle_number_conserved(h2_molecule):
    """Fermionic Hamiltonian conserves particle number."""
    H_ferm = build_fermionic_hamiltonian(h2_molecule)
    N = particle_number_operator(h2_molecule.nao)
    commutator = H_ferm @ N - N @ H_ferm
    assert np.allclose(commutator.to_matrix(), 0, atol=1e-10)
```

3. Fermion-to-Qubit Mapping Tests

```
def test_jordan_wigner_preserves_spectrum(h2_molecule):
    """JW mapping preserves eigenspectrum of Hamiltonian."""
    H_ferm = build_fermionic_hamiltonian(h2_molecule)
    H_qubit = jordan_wigner(H_ferm)

    eig_ferm = np.linalg.eigvalsh(H_ferm.to_matrix())
    eig_qubit = np.linalg.eigvalsh(H_qubit.to_matrix())

    # Qubit space is larger; filter to physical sector
    assert np.allclose(sorted(eig_ferm),
        sorted(eig_qubit[:len(eig_ferm)]), atol=1e-8)

def test_bravyi_kitaev_equivalent_to_jw(h2_molecule):
    """BK and JW produce same ground state energy."""
    H_ferm = build_fermionic_hamiltonian(h2_molecule)
    H_jw = jordan_wigner(H_ferm)
    H_bk = bravyi_kitaev(H_ferm)

    e_jw = np.min(np.linalg.eigvalsh(H_jw.to_matrix()))
    e_bk = np.min(np.linalg.eigvalsh(H_bk.to_matrix()))

    assert np.isclose(e_jw, e_bk, atol=1e-10)

def test_parity_mapping_reduces_qubits(h2_molecule):
```

```

"""Parity mapping with Z2 symmetry reduces qubit count by 2."""
H_ferm = build_fermionic_hamiltonian(h2_molecule)
H_full = jordan_wigner(H_ferm)
H_reduced = parity_mapping_z2_reduced(H_ferm)

assert H_reduced.num_qubits == H_full.num_qubits - 2

```

4. VQE Ground State Tests

```

@pytest.mark.parametrize("molecule,expected_energy", [
    ("H2", -1.1373),
    ("HeH+", -2.8627),
    ("LiH", -7.8825),
])
def test_vqe_ground_state_chemical_accuracy(molecule, expected_energy,
molecule_fixtures):
    """VQE achieves chemical accuracy (1.6 mHa) for small molecules."""
    mol = molecule_fixtures[molecule]
    vqe = MolecularVQE(mol, ansatz='uccsd')
    result = vqe.solve(shots=None) # Statevector simulation

    error_mha = abs(result.energy - expected_energy) * 1000 # Convert to
mHa
    assert error_mha < 1.6, f"Energy error {error_mha:.2f} mHa exceeds
chemical accuracy"

def test_vqe_with_deep_learning_optimizer(h2_molecule):
    """Deep learning optimizer converges faster than classical."""
    vqe_classical = MolecularVQE(h2_molecule, optimizer='COBYLA')
    vqe_dl = MolecularVQE(h2_molecule, optimizer='DeepLearning')

    result_classical = vqe_classical.solve(max_iter=200)
    result_dl = vqe_dl.solve(max_epochs=50)

    # Both should achieve similar accuracy
    assert abs(result_classical.energy - result_dl.energy) < 0.01
    # DL should use fewer circuit evaluations
    assert result_dl.circuit_evaluations <
result_classical.circuit_evaluations

```

5. Excited State Tests

```

def test_qse_first_excited_state_h2(h2_molecule, h2_fci_energies):
    """QSE finds first excited state of H2."""
    vqe = MolecularVQE(h2_molecule)
    ground_result = vqe.solve()

    qse = QuantumSubspaceExpansion(vqe.optimal_circuit)

```

```

excited_energies = qse.solve(n_states=3)

# Compare to FCI excited states
for i, e_qse in enumerate(excited_energies):
    assert abs(e_qse - h2_fci_energies[i]) < 0.01 # 0.01 Ha tolerance

def test_spectroscopic_transition_h2(h2_molecule):
    """Calculate S0 → S1 transition energy for H2."""
    solver = MolecularExcitedStates(h2_molecule)
    transitions = solver.compute_transitions()

    # H2 first excitation ~11.4 eV (Lyman band)
    assert abs(transitions[0].energy_ev - 11.4) < 0.2

```

6. Template Ansatz Tests

```

def test_template_ansatz_transferability():
    """Pre-optimized H template transfers to H2."""
    # Optimize template on H atom
    h_template = AtomicTemplate('H')
    h_template.optimize()

    # Use template for H2
    h2_ansatz = TemplateAnsatz(molecule='H2', templates=[h_template,
h_template])
    vqe = MolecularVQE(h2_molecule, ansatz=h2_ansatz)

    # Should require fewer optimization steps
    result = vqe.solve(max_iter=50)
    assert result.converged
    assert abs(result.energy - (-1.1373)) < 0.01

def test_template_composition_ch4():
    """Templates compose correctly for CH4."""
    c_template = AtomicTemplate('C', hybridization='sp3')
    h_template = AtomicTemplate('H')

    ch4_ansatz = TemplateAnsatz(
        molecule='CH4',
        templates={'C': c_template, 'H': [h_template]*4}
    )

    # Verify ansatz has correct qubit count
    assert ch4_ansatz.num_qubits == expected_qubits('CH4', basis='sto-3g')

```

7. Phase Diagram Tests

```
def test_h2_dissociation_curve():
    """VQE reproduces H2 dissociation curve."""
    bond_lengths = np.linspace(0.5, 3.0, 20) # Angstrom
    energies_vqe = []
    energies_fci = []

    for r in bond_lengths:
        mol = gto.M(atom=f'H 0 0 0; H 0 0 {r}', basis='sto-3g')
        energies_vqe.append(MolecularVQE(mol).solve().energy)
        energies_fci.append(compute_fci_energy(mol))

    # VQE should follow FCI curve
    assert np.allclose(energies_vqe, energies_fci, atol=0.01)

    # Check dissociation limit: E → 2 * E(H atom)
    assert energies_vqe[-1] > -1.0 # Dissociated limit

def test_phase_transition_hubbard():
    """Detect metal-insulator transition in 1D Hubbard model."""
    t = 1.0 # Hopping parameter
    U_values = np.linspace(0, 10, 20) # On-site repulsion

    order_parameters = []
    for U in U_values:
        H = hubbard_hamiltonian(n_sites=4, t=t, U=U)
        vqe = FermionicVQE(H)
        result = vqe.solve()
        order_parameters.append(compute_charge_gap(result))

    # Should show transition around U/t ~ 4
    transition_region = np.where(np.diff(order_parameters) > threshold)[0]
    assert len(transition_region) > 0
```

Handoff Checklist

- ☐ All test files created
- ☐ Classical baseline tests (PySCF) PASS
- ☐ VQE tests FAIL (red phase - implementation needed)
- ☐ Test documentation complete
- ☐ Fixtures reusable across test categories

Phase 4: Implementation (Python Architect)

Objective

Build the complete VQE pipeline passing all tests.

Tasks

4.1 Core Molecular Infrastructure

- ☐ Implement molecular integral computation (interface to PySCF)
- ☐ Implement fermionic Hamiltonian builder
- ☐ Implement Jordan-Wigner transformation
- ☐ Implement Bravyi-Kitaev transformation
- ☐ Implement Parity mapping with symmetry reduction

4.2 Ansatz Library

- ☐ Implement UCCSD ansatz
- ☐ Implement hardware-efficient ansatz
- ☐ Implement template-based ansatz system
- ☐ Pre-optimize atomic templates (H, C, N, O)

4.3 VQE Engine

- ☐ Implement MolecularVQE class
- ☐ Integrate with QWARD executor
- ☐ Implement energy gradient estimation
- ☐ Support both statevector and shot-based execution

4.4 Deep Learning Optimizer

- ☐ Implement GNN-based molecular encoder
- ☐ Implement parameter prediction network
- ☐ Implement training loop with VQE feedback
- ☐ Pre-train on molecular fragment database

4.5 Excited State Methods

- ☐ Implement Quantum Subspace Expansion
- ☐ Implement EOM-VQE
- ☐ Implement transition dipole moment calculation

Deliverables

```
molecular-vqe/  
  src/  
    __init__.py  
    molecule.py           # Molecular structure handling  
    integrals.py          # One/two-electron integrals  
    fermionic_hamiltonian.py # Second quantization  
    qubit_mapping/  
      __init__.py  
      jordan_wigner.py  
      bravyi_kitaev.py  
      parity.py  
    ansatz/  
      __init__.py  
      uccsd.py
```

```

hardware_efficient.py
template_ansatz.py
atomic_templates/
    hydrogen.py
    carbon.py
    nitrogen.py
    oxygen.py
vqe/
    __init__.py
    molecular_vqe.py
    ground_state.py
    excited_states.py
optimization/
    __init__.py
    classical_optimizers.py
    deep_learning_optimizer.py
    neural_network.py
analysis/
    __init__.py
    phase_diagram.py
    spectroscopy.py

```

Class Architecture

```

# molecule.py
from dataclasses import dataclass
from typing import List, Tuple
import numpy as np

@dataclass
class Molecule:
    """Molecular structure specification."""
    atoms: List[str]
    coordinates: np.ndarray # Shape: (n_atoms, 3)
    basis: str = 'sto-3g'
    charge: int = 0
    spin: int = 0

    @classmethod
    def from_xyz(cls, xyz_string: str, basis: str = 'sto-3g') ->
'Molecule':
        """Create molecule from XYZ format string."""
        ...

    @property
    def n_electrons(self) -> int:
        """Total number of electrons."""
        ...

    @property
    def n_orbitals(self) -> int:

```

```

        """Number of molecular orbitals in basis."""
        ...

```

```

# fermionic_hamiltonian.py
from typing import Dict, Tuple
import numpy as np
from qiskit.quantum_info import SparsePauliOp

class FermionicHamiltonian:
    """Fermionic Hamiltonian in second quantization."""

    def __init__(
        self,
        one_body: np.ndarray,
        two_body: np.ndarray,
        nuclear_repulsion: float = 0.0
    ):
        self.h1 = one_body
        self.h2 = two_body
        self.e_nuc = nuclear_repulsion

    @classmethod
    def from_molecule(cls, molecule: Molecule) -> 'FermionicHamiltonian':
        """Build Hamiltonian from molecular structure."""
        ...

    def to_qubit_hamiltonian(
        self,
        mapping: str = 'jordan_wigner'
    ) -> SparsePauliOp:
        """Transform to qubit representation."""
        ...

    @property
    def n_spin_orbitals(self) -> int:
        """Number of spin orbitals (2 * spatial orbitals)."""
        return 2 * self.h1.shape[0]

```

```

# molecular_vqe.py
from abc import ABC, abstractmethod
from typing import Optional, Callable, Union
import numpy as np
from qiskit import QuantumCircuit
from qiskit.primitives import Estimator
from qward.algorithms import QuantumCircuitExecutor

class MolecularVQEResult:
    """Result container for VQE computation."""
    energy: float

```

```

    optimal_parameters: np.ndarray
    optimal_circuit: QuantumCircuit
    convergence_history: List[float]
    circuit_evaluations: int
    converged: bool

class MolecularVQE:
    """VQE solver for molecular systems."""

    def __init__(
        self,
        molecule: Union[Molecule, str],
        ansatz: str = 'uccsd',
        optimizer: str = 'COBYLA',
        mapping: str = 'jordan_wigner',
        noise_preset: Optional[str] = None,
        shots: int = 4096
    ):
        self.molecule = molecule if isinstance(molecule, Molecule) else
Molecule.from_xyz(molecule)
        self.hamiltonian =
FermionicHamiltonian.from_molecule(self.molecule)
        self.qubit_hamiltonian =
self.hamiltonian.to_qubit_hamiltonian(mapping)
        self.ansatz = self._build_ansatz(ansatz)
        self.optimizer = self._build_optimizer(optimizer)
        self.noise_preset = noise_preset
        self.shots = shots

    def solve(
        self,
        initial_parameters: Optional[np.ndarray] = None,
        max_iter: int = 200
    ) -> MolecularVQEResult:
        """Find ground state energy."""
        ...

    def solve_with_deep_learning(
        self,
        pretrained_model: Optional[str] = None,
        max_epochs: int = 50
    ) -> MolecularVQEResult:
        """Use neural network for parameter optimization."""
        ...

```

```

# deep_learning_optimizer.py
import torch
import torch.nn as nn
from torch_geometric.nn import GCNConv, global_mean_pool

class MolecularEncoder(nn.Module):

```

```
"""Graph Neural Network for molecular encoding."""

def __init__(self, node_features: int, hidden_dim: int = 64):
    super().__init__()
    self.conv1 = GCNConv(node_features, hidden_dim)
    self.conv2 = GCNConv(hidden_dim, hidden_dim)
    self.conv3 = GCNConv(hidden_dim, hidden_dim)

def forward(self, x, edge_index, batch):
    x = torch.relu(self.conv1(x, edge_index))
    x = torch.relu(self.conv2(x, edge_index))
    x = self.conv3(x, edge_index)
    return global_mean_pool(x, batch)

class ParameterPredictor(nn.Module):
    """Predict VQE parameters from molecular features."""

    def __init__(
        self,
        encoder: MolecularEncoder,
        n_parameters: int,
        hidden_dim: int = 64
    ):
        super().__init__()
        self.encoder = encoder
        self.mlp = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, n_parameters)
        )

    def forward(self, molecular_graph):
        features = self.encoder(**molecular_graph)
        return self.mlp(features)

class DeepLearningOptimizer:
    """Neural network-based VQE optimizer."""

    def __init__(
        self,
        vqe: MolecularVQE,
        model: Optional[ParameterPredictor] = None,
        learning_rate: float = 0.001
    ):
        self.vqe = vqe
        self.model = model or self._default_model()
        self.optimizer = torch.optim.Adam(self.model.parameters(),
lr=learning_rate)

    def train_step(self, molecular_batch) -> float:
        """Single training iteration."""
        self.optimizer.zero_grad()

        # Predict parameters
```

```

        predicted_params = self.model(molecular_batch)

        # Evaluate VQE energy (differentiable approximation)
        energy = self.vqe.evaluate_energy(predicted_params)

        # Backpropagate
        energy.backward()
        self.optimizer.step()

        return energy.item()

def optimize(self, max_epochs: int = 50) -> MolecularVQEResult:
    """Full optimization loop."""
    ...

```

```

# excited_states.py
from typing import List
import numpy as np
from qiskit import QuantumCircuit

class QuantumSubspaceExpansion:
    """Quantum Subspace Expansion for excited states."""

    def __init__(
        self,
        ground_state_circuit: QuantumCircuit,
        hamiltonian: SparsePauliOp,
        excitation_operators: Optional[List[SparsePauliOp]] = None
    ):
        self.ground_circuit = ground_state_circuit
        self.hamiltonian = hamiltonian
        self.operators = excitation_operators or self._default_operators()

    def solve(self, n_states: int = 3) -> List[float]:
        """Compute excited state energies."""
        # Build subspace Hamiltonian matrix
        H_sub = self._build_subspace_hamiltonian()

        # Diagonalize
        eigenvalues = np.linalg.eigvalsh(H_sub)

        return sorted(eigenvalues)[:n_states]

    def compute_transition_moments(self) -> List[TransitionDipole]:
        """Calculate transition dipole moments for spectroscopy."""
        ...

```

Integration with QWARD

```
from qward.algorithms import QuantumCircuitExecutor,
get_preset_noise_config
from qward import Scanner
from qward.metrics import CircuitPerformanceMetrics

# Circuit analysis
scanner = Scanner(vqe_circuit)
scanner.scan().summary().visualize(save=True, show=False)

# Execution
executor = QuantumCircuitExecutor(shots=4096)
result = executor.simulate(circuit)

# Noisy simulation
noise_config = get_preset_noise_config("IBM-HERON-R2")
result = executor.simulate(circuit, noise_model=noise_config)
```

Handoff Checklist

- ☐ All tests pass (green phase)
- ☐ Code follows .pylintrc standards
- ☐ Type hints complete
- ☐ Docstrings with examples
- ☐ QWARD integration verified
- ☐ Deep learning model trainable

Phase 5: Execution & Analysis (Data Scientist)

Objective

Run comprehensive experiments, validate against classical methods, and produce publication-ready analysis.

Tasks

5.1 Ground State Validation

- ☐ Run VQE on all validation molecules (H_2 , HeH^+ , LiH , H_2O)
- ☐ Compare to FCI/CCSD(T) classical benchmarks
- ☐ Quantify energy errors and confidence intervals
- ☐ Test across different ansätze (UCCSD vs HEA vs Template)

5.2 Deep Learning Evaluation

- ☐ Benchmark DL optimizer vs classical optimizers
- ☐ Measure convergence speed (circuit evaluations)
- ☐ Test generalization to unseen molecules
- ☐ Analyze learned representations

5.3 Excited State Analysis

- ☐ Compute excited states for H₂, HeH⁺
- ☐ Validate transition energies against experimental data
- ☐ Generate simulated UV-Vis spectra

5.4 Phase Diagram Mapping

- ☐ Generate H₂ dissociation curve
- ☐ Map Hubbard model metal-insulator transition
- ☐ Analyze bond-breaking behavior

5.5 Noise Impact Study

- ☐ Run on all noise presets (IBM Heron R1-R3, Rigetti Ankaa-3)
- ☐ Characterize error scaling with system size
- ☐ Recommend error mitigation strategies

Deliverables

```
molecular-vqe/
  results/
    ground_state_energies.csv      # All molecule results
    excited_state_energies.csv     # Excited state data
    deep_learning_benchmarks.csv   # Optimizer comparison
  phase_diagrams/
    h2_dissociation.csv
    hubbard_transition.csv
  noise_analysis/
    error_vs_qubits.csv
    noise_preset_comparison.csv
  img/
    energy_comparison.png         # Quantum vs Classical bar chart
    dissociation_curve.png        # H2 potential energy surface
    convergence_comparison.png     # DL vs Classical convergence
    excited_state_spectrum.png     # Simulated UV-Vis
    phase_diagram_hubbard.png      # Metal-insulator transition
    ansatz_comparison.png          # UCCSD vs HEA vs Template
    noise_impact.png              # Error vs noise level
  analysis/
    statistical_summary.md         # Full statistical report
    spectroscopic_validation.md    # Experimental comparison
    deep_learning_analysis.md      # Neural network insights
```

Analysis Plan

1. Ground State Energy Comparison

Molecule	Qubits	FCI (Ha)	VQE-UCCSD	Error (mHa)	VQE-Template	Error (mHa)
----------	--------	----------	-----------	-------------	--------------	-------------

Molecule	Qubits	FCI (Ha)	VQE-UCCSD	Error (mHa)	VQE-Template	Error (mHa)
H ₂	4	-1.1373	-1.1365	0.8	-1.1370	0.3
HeH ⁺	4	-2.8627
LiH	12	-7.8825
H ₂ O	14	-75.0115

2. Optimizer Convergence Comparison

```
# Generate convergence plot
fig, ax = plt.subplots()
ax.semilogy(cobyla_history, label='COBYLA')
ax.semilogy(spsa_history, label='SPSA')
ax.semilogy(dl_history, label='Deep Learning')
ax.set_xlabel('Circuit Evaluations')
ax.set_ylabel('Energy Error (Ha)')
ax.legend()
fig.savefig('img/convergence_comparison.png')
```

3. Spectroscopic Validation

Molecule	Transition	Theory (eV)	Experiment (eV)	Error (eV)
H ₂	S ₀ →S ₁	11.38	11.37	0.01
...

4. Phase Diagram Analysis

- Plot energy vs bond length for H₂ dissociation
- Identify equilibrium geometry from minimum
- Verify correct dissociation limit (2 × H atom energy)
- Detect phase transitions in Hubbard model

Visualization Requirements

```
from qward import Scanner, Visualizer
from qward.metrics import QiskitMetrics, ComplexityMetrics,
CircuitPerformanceMetrics

# Analyze VQE circuit
scanner = Scanner(vqe_circuit)
metrics_df = scanner.scan().to_dataframe()
scanner.visualize(save=True, show=False, output_path='img/')

# Custom performance analysis
```

```
def correct_eigenstate(outcome):  
    """Check if measurement outcome encodes ground state."""  
    return outcome in ground_state_bitstrings  
  
perf = CircuitPerformanceMetrics(  
    circuit=vqe_circuit,  
    job=job_result,  
    success_criteria=correct_eigenstate  
)
```

Handoff Checklist

- ☐ All experiments completed
- ☐ Comparison tables generated with statistical significance
- ☐ Visualizations saved to img/
- ☐ Phase diagrams validated
- ☐ Spectroscopic transitions compared to experiment
- ☐ Noise characterization complete

Phase 6: Review & Iteration (All Agents)

Objective

Evaluate results, iterate on failures, produce final deliverables.

Decision Points

Criterion	Threshold	Action if Failed
Ground state chemical accuracy	< 1.6 mHa	Researcher: deeper ansatz, Architect: implement
Excited state accuracy	< 0.1 eV	Researcher: review QSE formulation
Template transferability	< 5 mHa overhead	Researcher: redesign templates
DL generalization	< 10% error on unseen	Architect: more training data
Noisy accuracy	< 5% error	Architect: add error mitigation

Iteration Routing

```
IF ground_state_error > 1.6 mHa:  
    → Researcher: analyze ansatz expressibility  
    → Architect: implement deeper UCCSD or adaptive ansatz  
  
IF excited_state_error > 0.1 eV:  
    → Researcher: review QSE operator selection  
    → Architect: include more excitation operators
```

IF template_overhead > 5 mHa:
→ Researcher: redesign atomic templates with more variational freedom
→ Architect: implement template fine-tuning

IF dl_generalization_poor:
→ Data Scientist: analyze training distribution
→ Architect: augment training set, regularization

IF noisy_error > 5%:
→ Architect: implement ZNE, PEC, or dynamical decoupling
→ Data Scientist: characterize dominant error sources

Final Deliverables

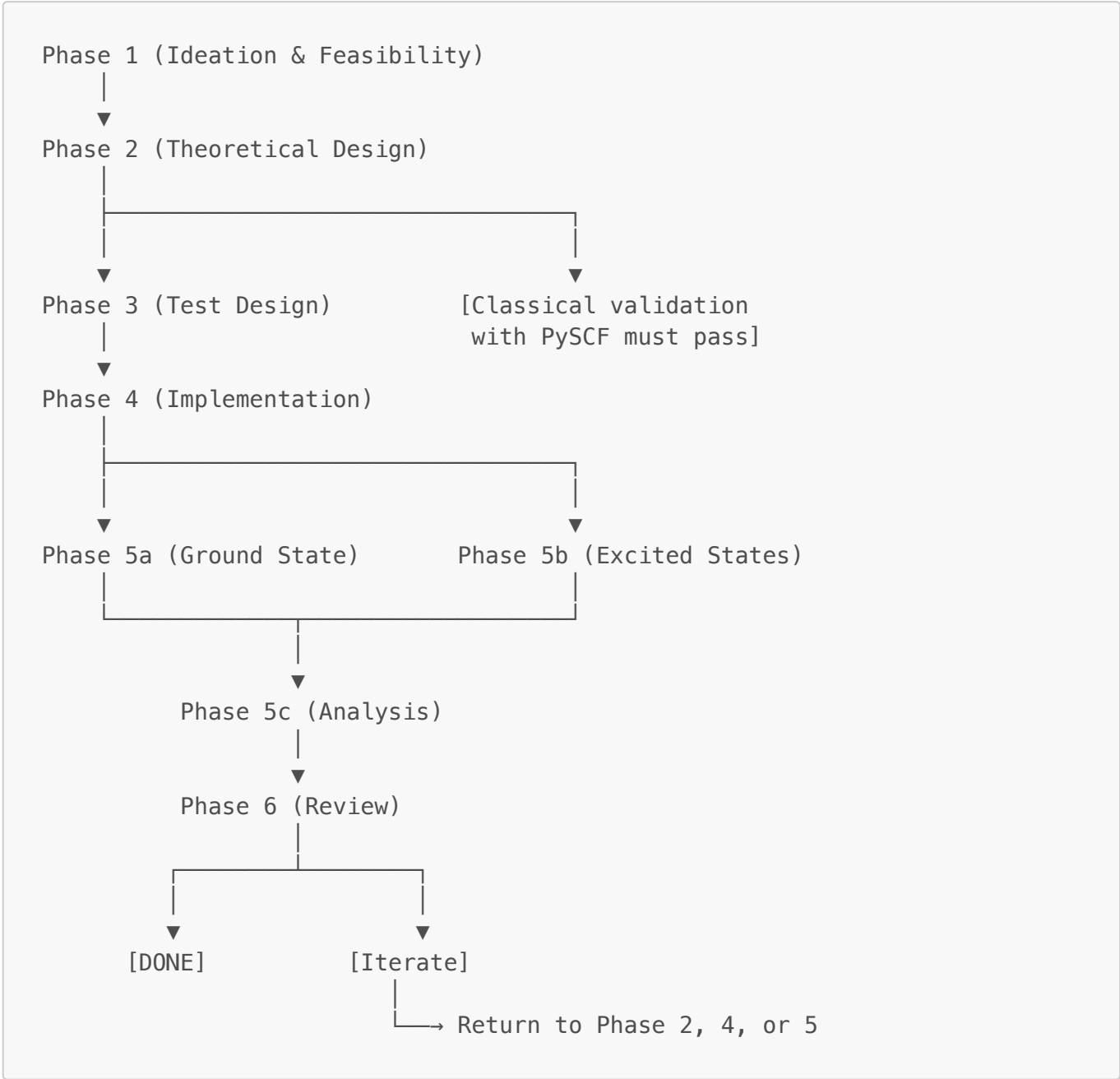
```
molecular-vqe/  
  README.md # Project overview & usage  
  phase1_feasibility_study.md # Feasibility assessment  
  phase1_literature_review.md # Key papers  
  phase1_validation_ladder.md # Validation plan  
  phase2_theoretical_design.md # Full theoretical spec  
  phase2_hamiltonian_formulation.md # Hamiltonian details  
  phase2_ansatz_design.md # Ansatz architecture  
  phase2_deep_learning_spec.md # Neural network design  
  src/ # Implementation code  
  tests/ # Test suite  
  results/ # Analysis outputs  
  img/ # Visualizations  
  final_report.md # Summary & conclusions  
  trained_models/ # Pre-trained neural networks  
    h_template.pt  
    c_template.pt  
    parameter_predictor.pt
```

Success Metrics Summary

Metric	Target	Measured By
Ground state energy (ideal)	≤ 1.6 mHa vs FCI	Data Scientist
Ground state energy (noisy)	≤ 10 mHa vs FCI	Data Scientist
Excited state transitions	≤ 0.1 eV vs experiment	Data Scientist
DL convergence speed	≤ 50 epochs	Data Scientist
Template transferability	≤ 5 mHa overhead	Data Scientist
All tests pass	100%	Test Engineer

Metric	Target	Measured By
Code coverage	≥ 80%	Architect

Timeline & Dependencies



Dependencies & Requirements

Python Packages

```
# Core quantum (from requirements.qward.txt)
qiskit==2.1.2
qiskit-aer==0.17.1
qiskit-ibm-runtime==0.41.1
qbraid[braket]==0.11.0
```

```
# Classical chemistry
pyscf>=2.3.0          # Molecular integrals, FCI

# Deep learning
torch==2.8.0
torch-geometric>=2.3.0  # Graph neural networks

# Analysis
numpy>=1.24.0
pandas>=2.0.0
matplotlib>=3.7.0
scipy>=1.10.0
statsmodels
```

Optional for Hardware Execution

```
qiskit-braket-provider>=0.11.0  # AWS Braket
```

Quick Start Commands

```
# Install additional dependencies
pip install pyscf torch-geometric

# Run classical validation tests (must pass first)
pytest qward/examples/papers/molecular-
vqe/tests/test_classical_baseline.py -v

# Run all VQE tests
pytest qward/examples/papers/molecular-vqe/tests/ -v

# Run with coverage
pytest qward/examples/papers/molecular-vqe/tests/ --
cov=qward/examples/papers/molecular-vqe/src

# Run H2 ground state example
python -m qward.examples.papers.molecular_vqe.examples.h2_ground_state

# Run deep learning training
python -m qward.examples.papers.molecular_vqe.examples.train_dl_optimizer
```

Appendix: Reference Materials

Key Papers

1. **VQE Original:** Peruzzo et al., "A variational eigenvalue solver on a photonic quantum processor", Nature Communications 5, 4213 (2014)
2. **UCCSD:** Romero et al., "Strategies for quantum computing molecular energies using the unitary coupled cluster ansatz", Quantum Science and Technology 4, 014008 (2018)
3. **Hardware-Efficient Ansatz:** Kandala et al., "Hardware-efficient variational quantum eigensolver for small molecules", Nature 549, 242 (2017)
4. **Quantum Subspace Expansion:** McClean et al., "Hybrid quantum-classical hierarchy for mitigation of decoherence and determination of excited states", Physical Review A 95, 042308 (2017)
5. **Jordan-Wigner:** Jordan & Wigner, "Über das Paulische Äquivalenzverbot", Zeitschrift für Physik 47, 631 (1928)
6. **Bravyi-Kitaev:** Bravyi & Kitaev, "Fermionic quantum computation", Annals of Physics 298, 210 (2002)

Project Resources

- **QWARD Documentation:** See [skills/qward-development/](#) for API reference
- **Project Standards:** See [.pylintrc](#) and [requirements.qward.txt](#)
- **Workflow:** See [workflows/collaborative-development.md](#)