Quantum Circuit Simulator Documentation

Table of Contents

Overview

This quantum circuit simulator provides a comprehensive platform for simulating quantum algorithms and circuits using classical computers. Built with PyTorch, it offers memory-optimized operations for simulating quantum systems up to practical limits.

Key Features

- **N-qubit quantum state simulation** with memory-optimized linear operations
- **Comprehensive gate library** including Pauli gates, rotations, and multi-qubit gates
- **Advanced quantum algorithms**: VQE, QAOA, QFT, QPE
- **Rich visualization tools** for states, circuits, and results
- **Hardware-efficient ansätze** for variational algorithms

Quick Start Guide

Basic Setup

```python
```

```python
from simulator import QuantumSimulator
from classes.plotter import QuantumPlotter

# Create a 2-qubit simulator
sim = QuantumSimulator(n_qubits=2)
plotter = QuantumPlotter()

# Apply some gates
sim.h(0)       # Hadamard on qubit 0
sim.cnot(0, 1)  # CNOT with control=0, target=1

# Visualize the state
sim.print_state()
sim.print_probabilities()

# Measure the system
results = sim.measure(shots=1000)
print(f"Measurement results: {results[:10]}")  # First 10 results
```

Creating Entangled States

```python
# Bell state preparation
sim.reset()
sim.h(0)
sim.cnot(0, 1)
print("Bell state created!")
sim.print_state()  # Should show (|00⟩ + |11⟩)/√2

# GHZ state for multiple qubits
sim_3 = QuantumSimulator(n_qubits=3)
sim_3.create_ghz_state()
sim_3.print_state()  # (|000⟩ + |111⟩)/√2
```

Simulator vs Real Quantum Computers

What the Simulator Does Well

Aspect	Simulator	Real Quantum Computer
State Access	✅ Full quantum state vector available	❌ No direct state access
Perfect Gates	✅ Exact unitary operations	❌ Noisy, imperfect gates
Coherence	✅ Infinite coherence time	❌ Limited by decoherence (~100μs)
Measurement	✅ Exact probability sampling	❌ Measurement errors (~1-5%)
Reproducibility	✅ Deterministic results	❌ Statistical variations
Gate Fidelity	✅ 100% fidelity	❌ 99.9% single-qubit, 99% two-qubit

Key Differences

1. Noise and Errors

```python
# Simulator: Perfect operations
sim.h(0)  # Exact Hadamard gate

# Real quantum computer would have:
# - Gate errors (~0.1% single-qubit, ~1% two-qubit)
# - Decoherence during operation
# - Measurement errors (~1-5%)
# - Crosstalk between qubits
```

2. Scalability

```python
# Simulator: Limited by classical memory
# 20 qubits = 2^20 = 1M complex numbers ≈ 8MB
# 30 qubits = 2^30 = 1B complex numbers ≈ 8GB
# 40 qubits = 2^40 = 1T complex numbers ≈ 8TB

# Real quantum computer: Exponential advantage potential
# Can theoretically scale to thousands of qubits
# But currently limited by noise and coherence
```

3. Connectivity and Topology

```python
```

```
# Simulator: All-to-all connectivity
sim.cnot(0, 15)  # Any qubit can interact with any other

# Real quantum computer: Limited connectivity
# - Only nearest neighbors can directly interact
# - Requires SWAP gates for distant interactions
# - Circuit depth increases significantly
```

4. **Quantum Advantage**

The simulator **cannot** demonstrate quantum advantage because:

- It runs on classical computers
- Simulation time grows exponentially with qubit count
- Real quantum computers could potentially solve certain problems faster

When to Use Each

Use the Simulator for:

- Algorithm development and testing
- Educational purposes
- Small-scale proof-of-concepts (< 20 qubits)
- Perfect gate analysis
- State visualization and debugging

Use Real Quantum Computers for:

- Quantum advantage demonstrations
- NISQ algorithm research
- Large-scale problems (when noise-tolerant)
- Studying real quantum effects

Core Components

1. QuantumSimulator Class

The main simulation engine that maintains quantum state and applies operations.

```
python
```

```python
# Initialize with n qubits
sim = QuantumSimulator(n_qubits=4)

# State management
sim.reset()                # Reset to |0000⟩
sim.get_state()            # Get current state vector
sim.set_state(custom_state)  # Set custom state
sim.get_probabilities()    # Get |amplitude|² for all states
```

2. Gate Library

Comprehensive set of quantum gates with optimized implementations.

Single-Qubit Gates

```python
# Pauli gates
sim.x(0)   # Pauli-X (bit flip)
sim.y(0)   # Pauli-Y
sim.z(0)   # Pauli-Z (phase flip)

# Hadamard and phase gates
sim.h(0)   # Hadamard (superposition)
sim.s(0)   # S gate (π/2 phase)
sim.t(0)   # T gate (π/4 phase)

# Rotation gates
sim.rx(angle, 0)  # Rotation around X-axis
sim.ry(angle, 0)  # Rotation around Y-axis
sim.rz(angle, 0)  # Rotation around Z-axis
```

Multi-Qubit Gates

```python
# Two-qubit gates
sim.cnot(control=0, target=1)  # Controlled-NOT
sim.cz(control=0, target=1)    # Controlled-Z
```

3. Memory-Optimized Operations

The simulator uses linear algebra operations instead of full tensor products for efficiency:

```python
# Traditional approach: O(4^n) memory for n-qubit gates
# Our approach: O(2^n) memory with optimized indexing

# Example: For 10 qubits
# Traditional: 4^10 = 1M entries in gate matrix
# Our method: 2^10 = 1K entries in state vector operations
```

Quantum Algorithms

1. Variational Quantum Eigensolver (VQE)

VQE finds the ground state energy of quantum systems using a hybrid quantum-classical approach.

How it Works:

1. Prepare a parameterized quantum state $|\psi(\theta)\rangle$
2. Measure the expectation value $\langle\psi(\theta)|H|\psi(\theta)\rangle$
3. Classically optimize parameters θ to minimize energy
4. Repeat until convergence

```python
```

```python
from vqe.vqe import VariationalQuantumEigensolver
from vqe.hamiltonian import Hamiltonian, PauliString
from vqe.ansatz import HardwareEfficientAnsatz

# Define Hamiltonian: H = 0.5*Z₀ + 0.5*Z₁ - 0.25*Z₀Z₁
hamiltonian = Hamiltonian([
    PauliString(['Z'], 0.5, [0]),
    PauliString(['Z'], 0.5, [1]),
    PauliString(['Z', 'Z'], -0.25, [0, 1])
])

# Create ansatz
ansatz = HardwareEfficientAnsatz(n_qubits=2, depth=2)

# Run VQE
vqe = VariationalQuantumEigensolver(hamiltonian, ansatz, None)
initial_params = np.random.uniform(0, 2*np.pi, ansatz.n_parameters)
result = vqe.run(initial_params, n_qubits=2)

print(f"Ground state energy: {result['energy']:.6f}")
```

Applications:

- Quantum chemistry (molecular ground states)
- Materials science (electronic structure)
- Optimization problems
- Quantum machine learning

2. Quantum Approximate Optimization Algorithm (QAOA)

QAOA solves combinatorial optimization problems by encoding them as quantum Hamiltonians.

How it Works:

1. Encode optimization problem as cost Hamiltonian H_C
2. Prepare initial superposition state
3. Alternate between cost and mixer unitaries:
   - Cost layer: e^(-iγH_C)
   - Mixer layer: e^(-iβH_M)
4. Measure to find approximate solutions

```python
from vqe.qaoa import QuantumApproximateOptimization

# Define Max-Cut problem on a triangle graph
graph_edges = [(0, 1), (1, 2), (0, 2)]
n_qubits = 3

# Create enhanced QAOA instance
qaoa = QuantumApproximateOptimization(
    n_layers=2,
    mixer_type='standard',
    use_warm_start=True,
    adaptive_initialization=True
)

# Solve Max-Cut
result = qaoa.solve_max_cut(graph_edges, n_qubits, shots=1000)

print(f"Best cut size: {result['best_cut_size']}")
print(f"Best solution: {result['best_cut_solution']}")
print(f"Approximation ratio: {result['approximation_ratio']:.3f}")
```

Enhanced Features:

- **Multiple mixer types**: Standard, XY, ring, asymmetric
- **Warm start initialization**: Classical pre-solving
- **Adaptive parameter initialization**: Problem-aware starting points
- **Multi-trial optimization**: Increased success probability

3. Quantum Fourier Transform (QFT)

QFT is the quantum analog of the discrete Fourier transform, essential for many quantum algorithms.

How it Works:

1. Apply Hadamard gates with controlled rotations
2. Implements: $|j\rangle \rightarrow (1/\sqrt{N}) \Sigma_k e^{(2\pi ijk/N)} |k\rangle$
3. Used in period finding, phase estimation, and Shor's algorithm

```python
```

```python
from algos.qft import QuantumFourierTransform

sim = QuantumSimulator(n_qubits=3)
qft = QuantumFourierTransform(sim)

# Prepare initial state |101⟩
sim.x(0)
sim.x(2)

# Apply QFT to all qubits
qft.qft([0, 1, 2])

# The state is now the QFT of |101⟩
sim.print_state()

# Apply inverse QFT
qft.inverse_qft([0, 1, 2])
sim.print_state()  # Should return to |101⟩
```

4. Quantum Phase Estimation (QPE)

QPE estimates the eigenphase of a unitary operator, fundamental to many quantum algorithms.

How it Works:

1. Prepare eigenstate |ψ⟩ of unitary U: U|ψ⟩ = e^(iφ)|ψ⟩
2. Use controlled-U operations with counting qubits
3. Apply inverse QFT to extract phase φ
4. Measure counting qubits to get phase estimate

```python
```

```python
from algos.qpe import QuantumPhaseEstimation

sim = QuantumSimulator(n_qubits=5)  # 4 counting + 1 eigenstate
qpe = QuantumPhaseEstimation(sim)

# Estimate phase of RZ(π/4) gate
true_phase = np.pi/4
estimated_phase, results = qpe.estimate_eigenphase(
    phase=true_phase,
    n_counting_qubits=4
)

print(f"True phase: {true_phase:.6f}")
print(f"Estimated phase: {estimated_phase:.6f}")
print(f"Error: {abs(true_phase/(2*np.pi) - estimated_phase):.6f}")
```

Usage Examples

Example 1: Creating and Analyzing Bell States

```python
```

```python
from simulator import QuantumSimulator
from classes.plotter import QuantumPlotter

sim = QuantumSimulator(n_qubits=2)
plotter = QuantumPlotter()

# Create all four Bell states
bell_states = {
    'Φ+': lambda: [sim.h(0), sim.cnot(0, 1)],
    'Φ-': lambda: [sim.h(0), sim.z(0), sim.cnot(0, 1)],
    'Ψ+': lambda: [sim.h(0), sim.x(1), sim.cnot(0, 1)],
    'Ψ-': lambda: [sim.h(0), sim.x(1), sim.z(0), sim.cnot(0, 1)]
}

for name, preparation in bell_states.items():
    sim.reset()
    [gate() for gate in preparation()]
    print(f"\n{name} Bell state:")
    sim.print_state()

    # Plot state amplitudes
    fig = plotter.plot_state_amplitudes(sim)
    plotter.save_plot(fig, f'bell_state_{name.lower()}.png')
```

Example 2: Quantum Teleportation Protocol

```python
```

```
def quantum_teleportation_demo():
    sim = QuantumSimulator(n_qubits=3)
    # Qubits: 0=Alice's qubit, 1=Alice's ancilla, 2=Bob's qubit

    # Step 1: Prepare unknown state |ψ⟩ = α|0⟩ + β|1⟩
    # Let's use |ψ⟩ = cos(π/6)|0⟩ + sin(π/6)|1⟩
    sim.ry(np.pi/3, 0)  # RY(π/3) creates cos(π/6)|0⟩ + sin(π/6)|1⟩

    print("Initial state to teleport:")
    sim.print_state()

    # Step 2: Create Bell pair between Alice's ancilla and Bob's qubit
    sim.h(1)
    sim.cnot(1, 2)

    # Step 3: Bell measurement on Alice's qubits
    sim.cnot(0, 1)
    sim.h(0)

    # Step 4: Measure Alice's qubits (simulated)
    # In real protocol, Bob applies corrections based on measurement
    results = sim.measure_single_qubit(0, shots=1000)
    alice_0 = 1 if np.mean(results) > 0.5 else 0

    results = sim.measure_single_qubit(1, shots=1000)
    alice_1 = 1 if np.mean(results) > 0.5 else 1

    # Step 5: Bob's corrections (classically determined)
    if alice_1 == 1:
        sim.x(2)
    if alice_0 == 1:
        sim.z(2)

    print(f"\nAfter teleportation (Alice measured {alice_0}{alice_1}):")
    print("Bob's qubit should have the original state")

    # Extract Bob's qubit state (qubit 2)
    probs = sim.get_probabilities()
    # Bob's |0⟩ probability = prob(|x00⟩) + prob(|x01⟩) + prob(|x10⟩) + prob(|x11⟩) where x=0
    bob_prob_0 = sum([probs[i] for i in range(8) if (i >> 0) & 1 == 0])
    print(f"Bob's qubit |0⟩ probability: {bob_prob_0:.4f}")
```

Example 3: Grover's Search Algorithm

python

```
def grovers_search(target_item, n_qubits):
    """Grover's algorithm to find target_item in unsorted database"""
    sim = QuantumSimulator(n_qubits=n_qubits)

    # Number of iterations for optimal success probability
    N = 2**n_qubits
    iterations = int(np.pi * np.sqrt(N) / 4)

    # Step 1: Initialize superposition
    for qubit in range(n_qubits):
        sim.h(qubit)

    print(f"Searching for item {target_item} in {N} items")
    print(f"Optimal iterations: {iterations}")

    # Grover iterations
    for iteration in range(iterations):
        # Oracle: flip phase of target state
        # This is a simplified oracle - real implementation depends on problem
        if target_item < N:
            # Convert target to binary and apply conditional phase flip
            binary = f"{target_item:0{n_qubits}b}"

            # Apply X gates to qubits that should be |0⟩ in target state
            for i, bit in enumerate(binary):
                if bit == '0':
                    sim.x(i)

            # Multi-controlled Z gate (simplified for small systems)
            if n_qubits == 2:
                sim.cz(0, 1)
            elif n_qubits == 3:
                # Approximate multi-controlled Z
                sim.h(2)
                sim.cnot(0, 2)
                sim.cnot(1, 2)
                sim.rz(np.pi, 2)
                sim.cnot(1, 2)
                sim.cnot(0, 2)
                sim.h(2)

            # Undo X gates
            for i, bit in enumerate(binary):
```

```
            if bit == '0':
                sim.x(i)

        # Diffusion operator (inversion about average)
        for qubit in range(n_qubits):
            sim.h(qubit)
            sim.x(qubit)

        # Multi-controlled Z on |11...1⟩ state
        if n_qubits == 2:
            sim.cz(0, 1)
        elif n_qubits == 3:
            sim.h(2)
            sim.cnot(0, 2)
            sim.cnot(1, 2)
            sim.rz(np.pi, 2)
            sim.cnot(1, 2)
            sim.cnot(0, 2)
            sim.h(2)

        for qubit in range(n_qubits):
            sim.x(qubit)
            sim.h(qubit)

        print(f"After iteration {iteration + 1}:")
        probs = sim.get_probabilities()
        print(f"Target probability: {probs[target_item]:.4f}")

    # Measure results
    results = sim.measure(shots=1000)
    success_rate = sum(1 for r in results if int(r, 2) == target_item) / 1000
    print(f"\nSuccess rate: {success_rate:.3f}")

    return results

# Run Grover's search
results = grovers_search(target_item=3, n_qubits=3)
```

Visualization Tools

The `QuantumPlotter` class provides comprehensive visualization capabilities.

State Visualization

```python
plotter = QuantumPlotter()

# Plot state amplitudes (real and imaginary parts)
amp_fig = plotter.plot_state_amplitudes(sim)

# Plot measurement probabilities
prob_fig = plotter.plot_state_probabilities(sim)

# Plot Bloch sphere (for single qubit reduced state)
bloch_fig = plotter.plot_bloch_sphere_projection(sim, qubit=0)

# Plot entanglement measure (for 2-qubit systems)
entangle_fig = plotter.plot_entanglement_measure(sim)
```

Circuit Visualization

```python
# Create and visualize quantum circuit
sim.h(0)
sim.cnot(0, 1)
sim.rz(np.pi/4, 1)
sim.measure()

circuit_fig = plotter.plot_quantum_circuit(sim.gate_sequence, sim.n_qubits)
```

Algorithm-Specific Plots

```python
```

```python
# VQE convergence
vqe_plots = vqe.plot_results(result, sim, plotter)

# QAOA results with graph analysis
qaoa_plots = qaoa.plot_results(result, graph_edges, n_qubits, plotter)

# Measurement histograms
results = sim.measure(shots=1000)
hist_fig = plotter.plot_measurement_histogram(results)

# State evolution over time
states_history = [sim.get_state() for _ in range(num_steps)]
evolution_fig = plotter.plot_state_evolution(states_history)
```

Saving and Displaying Plots

```python
# Save plots
plotter.save_plot(circuit_fig, 'my_circuit.png', dpi=300)

# Display all plots
plotter.show_all_plots()
```

Advanced Features

1. Custom State Preparation

```python
```

```
# Prepare custom superposition states
import numpy as np

# Create |+⟩⊗|+⟩ state manually
plus_plus_state = torch.tensor([0.5, 0.5, 0.5, 0.5], dtype=torch.complex64)
sim.set_state(plus_plus_state)

# Verify normalization
from classes.utils import QuantumUtils
print(f"Is normalized: {QuantumUtils.is_normalized(sim.get_state())}")

# Calculate fidelity with target state
fidelity = QuantumUtils.state_fidelity(sim.get_state(), plus_plus_state)
print(f"Fidelity: {fidelity:.6f}")
```

2. Custom Hamiltonians

python

```
from vqe.hamiltonian import Hamiltonian, PauliString

# Heisenberg model: H = Σᵢ (XᵢXᵢ₊₁ + YᵢYᵢ₊₁ + ZᵢZᵢ₊₁)
def create_heisenberg_hamiltonian(n_qubits, J=1.0):
    terms = []
    for i in range(n_qubits - 1):
        terms.append(PauliString(['X', 'X'], J, [i, i+1]))
        terms.append(PauliString(['Y', 'Y'], J, [i, i+1]))
        terms.append(PauliString(['Z', 'Z'], J, [i, i+1]))
    return Hamiltonian(terms)

# Transverse field Ising model: H = -J Σᵢ ZᵢZᵢ₊₁ - h Σᵢ Xᵢ
def create_tfim_hamiltonian(n_qubits, J=1.0, h=0.5):
    terms = []
    # ZZ interactions
    for i in range(n_qubits - 1):
        terms.append(PauliString(['Z', 'Z'], -J, [i, i+1]))
    # Transverse field
    for i in range(n_qubits):
        terms.append(PauliString(['X'], -h, [i]))
    return Hamiltonian(terms)
```

3. Error Analysis and Benchmarking

python

```python
def benchmark_gate_fidelity():
    """Benchmark gate operations against analytical results"""
    sim = QuantumSimulator(n_qubits=1)

    # Test Hadamard gate
    sim.h(0)
    expected_state = torch.tensor([1/np.sqrt(2), 1/np.sqrt(2)], dtype=torch.complex64)
    actual_state = sim.get_state()

    fidelity = QuantumUtils.state_fidelity(actual_state, expected_state)
    print(f"Hadamard gate fidelity: {fidelity:.10f}")

    # Test rotation gates
    angles = np.linspace(0, 2*np.pi, 10)
    for angle in angles:
        sim.reset()
        sim.ry(angle, 0)

        # Expected state after RY(θ): cos(θ/2)|0⟩ + sin(θ/2)|1⟩
        expected = torch.tensor([np.cos(angle/2), np.sin(angle/2)], dtype=torch.complex64)
        actual = sim.get_state()

        fidelity = QuantumUtils.state_fidelity(actual, expected)
        if fidelity < 0.999999:
            print(f"Warning: RY({angle:.3f}) fidelity = {fidelity:.10f}")

benchmark_gate_fidelity()
```

4. Memory Usage Monitoring

python

```
def analyze_memory_usage():
    """Analyze memory usage for different qubit counts"""
    for n_qubits in range(1, 21):
        sim = QuantumSimulator(n_qubits=n_qubits)
        usage = sim.get_memory_usage()

        print(f"{n_qubits:2d} qubits: {usage['state_vector_mb']:.2f} MB "
              f"({usage['state_size']} complex numbers)")

        if usage['state_vector_mb'] > 1000:  # > 1 GB
            print(f"Warning: {n_qubits} qubits requires {usage['state_vector_mb']:.1f} MB")
            break

analyze_memory_usage()
```

Performance Considerations

Memory Scaling

- **State vector size**: 2^n complex numbers for n qubits
- **Memory requirement**: $2^n \times 8$ bytes (complex64)
- **Practical limits**: ~20-25 qubits on typical systems

Qubits	State Size	Memory	Feasibility
10	1,024	8 KB	✅ Instant
15	32,768	256 KB	✅ Fast
20	1,048,576	8 MB	✅ Good
25	33,554,432	256 MB	⚠️ Slow
30	1,073,741,824	8 GB	❌ Requires HPC

Optimization Tips

python

```python
# 1. Use appropriate data types
sim = QuantumSimulator(n_qubits=10, device='cpu')  # CPU is faster for small systems

# 2. Batch measurements
results = sim.measure(shots=10000)  # Better than 10k individual calls

# 3. Reuse simulator instances
sim.reset()  # Faster than creating new instance

# 4. Monitor memory usage
usage = sim.get_memory_usage()
if usage['state_vector_mb'] > 1000:
    print("Warning: High memory usage")
```

Algorithm-Specific Performance

```python
# VQE: Optimize shot count vs accuracy trade-off
# More shots = better energy estimates but slower
vqe_result = vqe.run(initial_params, n_qubits=4, shots=1000)

# QAOA: Use problem-aware initialization
qaoa = QuantumApproximateOptimization(
    n_layers=3,
    use_warm_start=True,    # Classical pre-solving
    adaptive_initialization=True  # Problem-aware parameters
)
```

API Reference

QuantumSimulator Class

Constructor

```python
QuantumSimulator(n_qubits: int = 2, device: str = 'cpu')
```

State Management

```python
```

```
reset()                          # Reset to |00...0⟩
get_state() -> torch.Tensor      # Get current state vector
set_state(state: torch.Tensor)   # Set custom state (normalized)
get_probabilities() -> torch.Tensor  # Get |ψᵢ|² for all basis states
```

Single-Qubit Gates

```
python

x(qubit: int)                # Pauli-X gate
y(qubit: int)                # Pauli-Y gate
z(qubit: int)                # Pauli-Z gate
h(qubit: int)                # Hadamard gate
s(qubit: int)                # S phase gate
t(qubit: int)                # T gate
rx(angle: float, qubit: int)     # X-rotation
ry(angle: float, qubit: int)     # Y-rotation
rz(angle: float, qubit: int)     # Z-rotation
```

Multi-Qubit Gates

```
python

cnot(control: int, target: int)  # Controlled-NOT
cz(control: int, target: int)    # Controlled-Z
```

Measurement

```
python

measure(shots: int = 1) -> List[str]           # Full system measurement
measure_single_qubit(qubit: int, shots: int = 1) -> List[int]  # Single qubit
```

Utility Methods

```
python

print_state(precision: int = 4)       # Print state in bra-ket notation
print_probabilities(precision: int = 4)  # Print measurement probabilities
get_circuit_depth() -> int            # Number of gates applied
get_memory_usage() -> dict            # Memory usage statistics
```

Predefined States

```python
create_superposition_all()          # |++...+⟩ state
create_bell_state(state_type: str = '00')  # Bell states (2-qubit only)
create_ghz_state()                  # GHZ state: (|00...0⟩ + |11...1⟩)/√2
```

Algorithm Classes

VQE

```python
from vqe.vqe import VariationalQuantumEigensolver
from vqe.hamiltonian import Hamiltonian, PauliString
from vqe.ansatz import HardwareEfficientAnsatz

vqe = VariationalQuantumEigensolver(hamiltonian, ansatz, optimizer)
result = vqe.run(initial_params, n_qubits, shots=1000)
```

QAOA

```python
from vqe.qaoa import QuantumApproximateOptimization

qaoa = QuantumApproximateOptimization(n_layers=2)
result = qaoa.solve_max_cut(graph_edges, n_qubits, shots=1000)
```

QFT

```python
from algos.qft import QuantumFourierTransform

qft = QuantumFourierTransform(simulator)
qft.qft(qubits=[0, 1, 2])          # Apply QFT
qft.inverse_qft(qubits=[0, 1, 2])  # Apply inverse QFT
```

QPE

```python
```

```python
from algos.qpe import QuantumPhaseEstimation

qpe = QuantumPhaseEstimation(simulator)
estimated_phase, results = qpe.estimate_eigenphase(
    phase=true_phase,
    n_counting_qubits=4
)
```

Visualization Classes

QuantumPlotter

```python
from classes.plotter import QuantumPlotter

plotter = QuantumPlotter()

# State visualization
plot_state_amplitudes(simulator, figsize=(12, 6))
plot_state_probabilities(simulator, figsize=(10, 6))
plot_bloch_sphere_projection(simulator, qubit=0, figsize=(8, 8))

# Circuit visualization
plot_quantum_circuit(gate_sequence, n_qubits, figsize=None)

# Results visualization
plot_measurement_histogram(results, figsize=(10, 6))
plot_state_evolution(states_history, labels=None, figsize=(12, 8))

# Algorithm-specific plots
plot_vqe_convergence(history, figsize=(10, 6))
plot_qaoa_results(solution_counts, graph_edges, n_qubits, figsize=(12, 8))

# Utility methods
show_all_plots()
save_plot(fig, filename, dpi=300)
```

Utility Classes

QuantumUtils

```python
```

```python
from classes.utils import QuantumUtils

# State operations
QuantumUtils.normalize_state(state)
QuantumUtils.is_normalized(state, tolerance=1e-10)
QuantumUtils.state_fidelity(state1, state2)
QuantumUtils.tensor_to_numpy(tensor)

# State creation
QuantumUtils.create_bell_state(state_type='00', n_qubits=2, device='cpu')
QuantumUtils.create_ghz_state(n_qubits, device='cpu')

# Formatting
QuantumUtils.format_complex(z, precision=4)
QuantumUtils.state_to_string(state, n_qubits, precision=4)
```

Troubleshooting and Common Issues

1. Memory Errors

```python
# Problem: Out of memory for large qubit counts
# Solution: Monitor memory usage and reduce qubit count
sim = QuantumSimulator(n_qubits=25)  # May fail
usage = sim.get_memory_usage()
if usage['state_vector_mb'] > 1000:  # > 1GB
    print("Consider reducing qubit count or using HPC resources")
```

2. Numerical Precision Issues

```python
# Problem: Small numerical errors accumulate
# Solution: Use appropriate precision and normalization
state = sim.get_state()
if not QuantumUtils.is_normalized(state, tolerance=1e-10):
    print("Warning: State not normalized, renormalizing...")
    sim.set_state(QuantumUtils.normalize_state(state))
```

3. Gate Sequence Issues

```python
```

```python
# Problem: Incorrect gate ordering or parameters
# Solution: Verify gate sequence and use print statements
sim.h(0)
sim.cnot(0, 1)
print(f"Applied {sim.get_circuit_depth()} gates")
print("Gate sequence:", sim.gate_sequence)
```

4. Measurement Statistical Errors

```python
# Problem: High variance in measurement results
# Solution: Increase shot count for better statistics
results_1k = sim.measure(shots=1000)
results_10k = sim.measure(shots=10000)

# Calculate standard error: σ/√n where σ ≈ √(p(1-p))
prob_estimate = results_1k.count('00') / 1000
std_error = np.sqrt(prob_estimate * (1 - prob_estimate) / 1000)
print(f"Estimated probability: {prob_estimate:.3f} ± {std_error:.3f}")
```

5. Algorithm Convergence Issues

```python
```

```python
# Problem: VQE or QAOA not converging
# Solutions:
# 1. Try different optimizers
from scipy.optimize import minimize
result = minimize(cost_function, initial_params, method='COBYLA')

# 2. Use multiple random starts
best_result = None
best_energy = float('inf')
for trial in range(10):
    initial_params = np.random.uniform(0, 2*np.pi, n_params)
    result = minimize(cost_function, initial_params)
    if result.fun < best_energy:
        best_energy = result.fun
        best_result = result

# 3. Adjust optimizer parameters
result = minimize(
    cost_function,
    initial_params,
    method='COBYLA',
    options={'maxiter': 500, 'tol': 1e-8}
)
```

Educational Examples and Exercises

Exercise 1: Quantum Interferometry

Goal: Understand quantum superposition and interference

python

```python
def quantum_interferometry_demo():
    """Demonstrate quantum interference with Mach-Zehnder interferometer"""
    sim = QuantumSimulator(n_qubits=1)

    print("Quantum Interferometry Demo")
    print("=" * 40)

    # Step 1: Create superposition (beam splitter)
    sim.h(0)
    print("After first beam splitter:")
    sim.print_state()

    # Step 2: Add relative phase (phase shifter)
    phases = [0, np.pi/4, np.pi/2, 3*np.pi/4, np.pi]

    for phase in phases:
        sim_temp = QuantumSimulator(n_qubits=1)
        sim_temp.h(0)  # Beam splitter
        sim_temp.rz(phase, 0)  # Phase shifter
        sim_temp.h(0)  # Second beam splitter

        prob_0 = sim_temp.get_probabilities()[0].item()
        prob_1 = sim_temp.get_probabilities()[1].item()

        print(f"Phase = {phase:.3f}π: P(|0⟩) = {prob_0:.3f}, P(|1⟩) = {prob_1:.3f}")

    print("\nObservation: Probability oscillates with phase - quantum interference!")

quantum_interferometry_demo()
```

Exercise 2: Quantum Random Walk

Goal: Explore quantum vs classical random walks

python

```python
def quantum_random_walk(steps=4):
    """Compare quantum and classical random walks"""
    # Position qubits (log₂(2*steps+1) qubits needed)
    n_position_qubits = int(np.ceil(np.log2(2*steps + 1)))
    n_qubits = n_position_qubits + 1  # +1 for coin qubit

    sim = QuantumSimulator(n_qubits=n_qubits)

    print(f"Quantum Random Walk - {steps} steps")
    print(f"Using {n_qubits} qubits ({n_position_qubits} position + 1 coin)")

    # Initialize: |position=center,coin=+⟩
    center = steps  # Center position
    sim.h(0)  # Coin in superposition

    # Encode initial position (simplified for demo)
    # In practice, need more sophisticated position encoding

    for step in range(steps):
        # Quantum coin flip
        sim.h(0)

        # Conditional move (simplified)
        # Real implementation needs quantum arithmetic
        sim.cnot(0, 1)  # Move right if coin is |1⟩

        print(f"Step {step + 1}:")
        sim.print_probabilities()

    # Compare with classical random walk probabilities
    classical_probs = []
    for pos in range(-steps, steps + 1):
        # Binomial distribution for classical walk
        n_right = (pos + steps) // 2
        if 2*n_right - steps == pos and 0 <= n_right <= steps:
            prob = (1/2)**steps * np.math.comb(steps, n_right)
            classical_probs.append(prob)
        else:
            classical_probs.append(0)

    print(f"\nClassical probabilities: {classical_probs}")
    print("Quantum walks show different spreading behavior!")
```

```
quantum_random_walk(steps=3)
```

Exercise 3: Quantum Error Detection

Goal: Understand quantum error correction basics

```
python
```

```
def three_qubit_error_correction_demo():
    """Demonstrate 3-qubit bit-flip error correction"""
    sim = QuantumSimulator(n_qubits=3)

    print("3-Qubit Error Correction Demo")
    print("=" * 35)

    # Step 1: Prepare logical |0⟩ = |000⟩
    print("Logical |0⟩ encoded as |000⟩:")
    sim.print_state()

    # Step 2: Prepare logical |1⟩ = |111⟩
    sim.reset()
    sim.x(0)
    sim.x(1)
    sim.x(2)
    print("\nLogical |1⟩ encoded as |111⟩:")
    sim.print_state()

    # Step 3: Prepare logical |+⟩ = (|000⟩ + |111⟩)/√2
    sim.reset()
    sim.h(0)
    sim.cnot(0, 1)
    sim.cnot(0, 2)
    print("\nLogical |+⟩ = (|000⟩ + |111⟩)/√2:")
    sim.print_state()

    # Step 4: Introduce bit-flip error on qubit 1
    sim.x(1)  # Error!
    print("\nAfter bit-flip error on qubit 1:")
    sim.print_state()  # Should show (|010⟩ + |101⟩)/√2

    # Step 5: Error detection (measure syndrome)
    # Parity check 1: qubits 0 and 1
    # Parity check 2: qubits 1 and 2
    # (Simplified - real implementation needs ancilla qubits)

    print("\nError detected! Syndrome indicates error on qubit 1")

    # Step 6: Error correction
    sim.x(1)  # Correct the error
    print("\nAfter correction:")
    sim.print_state()  # Should return to (|000⟩ + |111⟩)/√2
```

```
    print("Original logical state recovered!")

three_qubit_error_correction_demo()
```

Research Applications and Extensions

1. Quantum Machine Learning

```
python
```

```python
def quantum_neural_network_demo():
    """Simple quantum neural network for binary classification"""
    from vqe.ansatz import HardwareEfficientAnsatz

    # Data: XOR problem
    X_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y_train = np.array([0, 1, 1, 0])  # XOR labels

    n_qubits = 2
    sim = QuantumSimulator(n_qubits=n_qubits)
    ansatz = HardwareEfficientAnsatz(n_qubits=n_qubits, depth=2)

    def encode_data(x):
        """Encode classical data into quantum state"""
        sim.reset()
        sim.ry(x[0] * np.pi, 0)
        sim.ry(x[1] * np.pi, 1)

    def quantum_neural_network(x, params):
        """QNN forward pass"""
        encode_data(x)
        ansatz.apply_circuit(sim, params)
        # Measure expectation value of Z₀
        prob_0 = sim.get_probabilities()[0] + sim.get_probabilities()[1]  # |0⟩ on qubit 0
        return 2 * prob_0 - 1  # Map [0,1] to [-1,1]

    def cost_function(params):
        """Mean squared error loss"""
        loss = 0
        for x, y_true in zip(X_train, y_train):
            y_pred = quantum_neural_network(x, params)
            y_target = 2 * y_true - 1  # Map {0,1} to {-1,1}
            loss += (y_pred - y_target)**2
        return loss / len(X_train)

    # Train the QNN
    from scipy.optimize import minimize
    initial_params = np.random.uniform(0, 2*np.pi, ansatz.n_parameters)

    result = minimize(cost_function, initial_params, method='COBYLA')

    print("Quantum Neural Network Training Results:")
    print(f"Final loss: {result.fun:.6f}")
```

```python
# Test predictions
for i, (x, y_true) in enumerate(zip(X_train, y_train)):
    y_pred = quantum_neural_network(x, result.x)
    y_pred_class = 1 if y_pred > 0 else 0
    print(f"Input {x}: True={y_true}, Pred={y_pred_class} (score={y_pred:.3f})")
```

2. Quantum Chemistry Simulation

python

```python
def h2_molecule_simulation():
    """Simulate H₂ molecule using VQE"""
    from vqe.vqe import VariationalQuantumEigensolver
    from vqe.hamiltonian import Hamiltonian, PauliString
    from vqe.ansatz import HardwareEfficientAnsatz

    # H₂ Hamiltonian in minimal basis (STO-3G)
    # H = -1.0523732 I + 0.39793742 Z₀ - 0.39793742 Z₁ - 0.01128010 Z₀Z₁ + 0.18093119 X₀X₁
    h2_terms = [
        PauliString([], -1.0523732, []),  # Identity term (constant)
        PauliString(['Z'], 0.39793742, [0]),
        PauliString(['Z'], -0.39793742, [1]),
        PauliString(['Z', 'Z'], -0.01128010, [0, 1]),
        PauliString(['X', 'X'], 0.18093119, [0, 1])
    ]

    h2_hamiltonian = Hamiltonian(h2_terms)

    # Use hardware-efficient ansatz
    ansatz = HardwareEfficientAnsatz(n_qubits=2, depth=2)

    # Run VQE
    vqe = VariationalQuantumEigensolver(h2_hamiltonian, ansatz, None)
    initial_params = np.random.uniform(0, 2*np.pi, ansatz.n_parameters)

    result = vqe.run(initial_params, n_qubits=2, shots=10000)

    print("H₂ Molecule VQE Simulation:")
    print(f"Ground state energy: {result['energy']:.6f} Hartree")
    print(f"Reference energy: -1.8572750 Hartree")  # Known exact result
    print(f"Error: {abs(result['energy'] - (-1.8572750)):.6f} Hartree")
    print(f"Converged: {result['converged']}")

    return result
```

3. Quantum Optimization Problems

python

```python
def traveling_salesman_qaoa():
    """Solve small TSP instance using QAOA"""
    # 3-city TSP with distance matrix
    distances = np.array([
        [0, 2, 3],
        [2, 0, 1],
        [3, 1, 0]
    ])

    n_cities = 3
    # Need n_cities² qubits for position encoding
    # Qubit (i,j) = 1 if city i is at position j in tour
    n_qubits = n_cities * n_cities

    print(f"TSP QAOA: {n_cities} cities, {n_qubits} qubits")
    print("Distance matrix:")
    print(distances)

    # Construct TSP Hamiltonian (simplified version)
    # Real TSP needs many constraint terms
    from vqe.hamiltonian import Hamiltonian, PauliString

    tsp_terms = []

    # Distance terms (simplified)
    for i in range(n_cities):
        for j in range(n_cities):
            for k in range(n_cities):
                if i != k:
                    # Cost for going from city i to city k
                    qubit1 = i * n_cities + j  # City i at position j
                    qubit2 = k * n_cities + ((j + 1) % n_cities)  # City k at next position

                    if qubit1 < n_qubits and qubit2 < n_qubits:
                        weight = distances[i, k] * 0.1  # Scale down
                        tsp_terms.append(PauliString(['Z', 'Z'], weight, [qubit1, qubit2]))

    if not tsp_terms:
        print("Warning: No valid TSP terms generated")
        return

    tsp_hamiltonian = Hamiltonian(tsp_terms)
```

```
    # Use QAOA
    from vqe.qaoa import QuantumApproximateOptimization
    qaoa = QuantumApproximateOptimization(n_layers=2)

    # Convert to Max-Cut format (simplified)
    graph_edges = [(term.qubits[0], term.qubits[1]) for term in tsp_terms if len(term.qubits) == 2]

    try:
        result = qaoa.solve_max_cut(graph_edges, n_qubits, shots=1000)
        print(f"Best solution: {result['best_cut_solution']}")
        print(f"Approximation ratio: {result['approximation_ratio']:.3f}")
    except Exception as e:
        print(f"QAOA failed: {e}")
        print("TSP is a complex problem requiring more sophisticated encoding")

# Note: This is a simplified demonstration
# Real TSP-QAOA requires careful constraint handling
```

Future Directions and Limitations

Current Limitations

1. **Exponential Memory Scaling**: Cannot simulate > ~25 qubits on typical hardware
2. **No Noise Modeling**: Perfect gates unlike real quantum devices
3. **Classical Simulation**: Cannot demonstrate true quantum advantage
4. **Limited Gate Set**: Basic gates only, no exotic operations

Possible Extensions

1. **Noise Models**: Add decoherence, gate errors, measurement errors
2. **Sparse State Representation**: For special quantum states
3. **GPU Acceleration**: Parallel operations for larger systems
4. **Quantum Error Correction**: Full stabilizer formalism
5. **Advanced Algorithms**: Shor's algorithm, HHL, quantum walks

Research Opportunities

- **NISQ Algorithm Development**: Test algorithms for near-term devices
- **Quantum Machine Learning**: Explore quantum advantage in ML
- **Quantum Chemistry**: Larger molecular systems

- **Optimization**: Novel quantum optimization approaches

Conclusion

This quantum circuit simulator provides a powerful platform for learning, researching, and developing quantum algorithms. While it cannot replace real quantum computers, it offers several advantages:

Educational Value:

- Complete state visibility for understanding
- Perfect operations for algorithm verification
- Rich visualization tools
- No hardware access needed

Research Applications:

- Algorithm prototyping and testing
- Parameter optimization for NISQ algorithms
- Quantum algorithm benchmarking
- Educational demonstrations

Best Practices:

- Start with small systems (< 10 qubits) for learning
- Use visualization tools extensively
- Compare with analytical results when possible
- Monitor memory usage for larger systems
- Consider noise effects when translating to real hardware

The simulator bridges the gap between quantum theory and implementation, making quantum computing accessible for education and research. As quantum hardware continues to improve, simulators like this will remain essential tools for algorithm development and verification.

Disclaimer: This simulator is for educational and research purposes. Results may not directly translate to real quantum hardware due to noise, connectivity constraints, and other practical limitations. Always validate results on actual quantum devices when possible.