Quantum Simulation Toolkit

User Guide and Reference Manual

Complete Guide to Hamiltonian Construction,
Time Evolution, and Density Matrix Operations

Generated: October 30, 2025

Table of Contents

1. Overview

This toolkit provides a comprehensive suite of Python modules for quantum simulation, including Hamiltonian construction from Pauli operators, time evolution using the Runge-Kutta method, and density matrix operations including partial traces.

Key Features:

• Intuitive Hamiltonian construction using string notation or programmatic interface
• 4th-order Runge-Kutta time evolution with adaptive step-size option
• Density matrix computation from state vectors: ρ = |ψ■■ψ|
• Partial trace operations for reduced density matrices
• Entanglement measures: purity, von Neumann entropy, fidelity
• Support for arbitrary qubit systems

Available Pauli Operators:

I = Identity, X = Pauli-X (bit flip), Y = Pauli-Y, Z = Pauli-Z (phase flip)

2. Module: hamiltonian_builder.py

2.1 String Notation Method

The simplest way to create Hamiltonians is using string notation. Write your Hamiltonian exactly as you would on paper, using ⊗ (or * or concatenation) for tensor products.

```
from hamiltonian_builder import hamiltonian_from_string

# Single term
H = hamiltonian_from_string("X⊗Z")

# Sum of terms
H = hamiltonian_from_string("X⊗I + I⊗X")

# With coefficients
H = hamiltonian_from_string("0.5*X⊗Z + 0.3*Z⊗X")

# Four qubits
H = hamiltonian_from_string("Z⊗I⊗X⊗Z + X⊗X⊗I⊗I")

# Compact notation (no symbols)
H = hamiltonian_from_string("XII + IXI + IIX")
```

2.2 HamiltonianBuilder Class

For systematic construction or when building Hamiltonians programmatically, use the HamiltonianBuilder class. This is especially useful for constructing many-body Hamiltonians with loops.

```
from hamiltonian_builder import HamiltonianBuilder

# Basic usage
builder = HamiltonianBuilder()
builder.add_term(['X', 'I', 'Z'], coefficient=1.0)
builder.add_term(['Z', 'X', 'I'], coefficient=1.0)
H = builder.build()

# Systematic construction with loop
n_qubits = 4
builder = HamiltonianBuilder()
for i in range(n_qubits - 1):
    ops = ['I'] * n_qubits
    ops[i] = 'X'
    ops[i+1] = 'X'
    builder.add_term(ops, coefficient=1.0)
H = builder.build()
```

2.3 Common Hamiltonian Examples

```
# Transverse-Field Ising Model
# H = -J*Σ(Z_i⊗Z_{i+1}) - h*Σ(X_i)
n = 3
J, h = 1.0, 0.5
builder = HamiltonianBuilder()
for i in range(n-1):
    ops = ['I'] * n
    ops[i], ops[i+1] = 'Z', 'Z'
    builder.add_term(ops, coefficient=-J)
for i in range(n):
    ops = ['I'] * n
    ops[i] = 'X'
    builder.add_term(ops, coefficient=-h)
H = builder.build()
```

3. Module: runge_kutta_hamiltonian.py

This module implements the 4th-order Runge-Kutta method for solving the time-dependent Schrödinger equation: i·dψ/dt = H|ψ■

3.1 Basic Time Evolution

```
import numpy as np
from runge_kutta_hamiltonian import runge_kutta_4
from hamiltonian_builder import hamiltonian_from_string

# Create Hamiltonian
H = hamiltonian_from_string("X⊗Z")

# Initial state |00■
psi0 = np.array([1, 0, 0, 0], dtype=complex)

# Evolve to time t
t = np.pi / 2
psi_final = runge_kutta_4(psi0, H, t, dt=0.001)

print("Final state:", psi_final)
print("Probabilities:", np.abs(psi_final)**2)
```

3.2 Function Parameters

Parameter	Type	Description
psi0	ndarray	Initial state vector at t=0 (complex)
H	ndarray	Hamiltonian matrix (Hermitian)
t	float	Final time to evolve to
dt	float	Time step (default: 0.001, smaller = more accurate)

4. Module: density_matrix_ops.py

This module provides functions for computing density matrices and partial traces, essential for analyzing entanglement and reduced quantum systems.

4.1 Computing Density Matrix

Convert a pure state |ψ■ to its density matrix representation ρ = |ψ■■ψ|:

```
from density_matrix_ops import state_to_density_matrix

# Pure state |+■ = (|0■ + |1■)/√2
psi = np.array([1, 1]) / np.sqrt(2)

# Compute density matrix
rho = state_to_density_matrix(psi)
print(rho)
# Output:
# [[0.5 0.5]
#  [0.5 0.5]]
```

4.2 Partial Trace

Compute reduced density matrices by tracing out subsystems:

```
from density_matrix_ops import partial_trace

# Bell state |Φ+■ = (|00■ + |11■)/√2
bell = np.array([1, 0, 0, 1]) / np.sqrt(2)
rho_full = state_to_density_matrix(bell)

# Trace out qubit 1, keep qubit 0
rho_A = partial_trace(rho_full, dims=[2, 2], trace_out=1)
print(rho_A)
# Output: maximally mixed state
# [[0.5 0. ]
#  [0.  0.5]]

# For 3 qubits, trace out qubits 1 and 2
rho_0 = partial_trace(rho, dims=[2, 2, 2], trace_out=[1, 2])
```

4.3 Entanglement Measures

```
from density_matrix_ops import purity, von_neumann_entropy

# Purity: Tr(ρ²)
P = purity(rho_A)  # = 1 for pure, < 1 for mixed

# Von Neumann entropy: -Tr(ρ log ρ)
S = von_neumann_entropy(rho_A)  # = 0 for pure, > 0 for entangled
```

```
# For Bell state: purity = 0.5, entropy = 1 bit
```

5. Complete Workflow Example

This example demonstrates the complete workflow: building a Hamiltonian, evolving a state, computing density matrices, and analyzing entanglement.

```
import numpy as np
from hamiltonian_builder import hamiltonian_from_string
from runge_kutta_hamiltonian import runge_kutta_4
from density_matrix_ops import (
    state_to_density_matrix,
    partial_trace,
    purity,
    von_neumann_entropy
)

# Step 1: Build Hamiltonian
H = hamiltonian_from_string("X⊗X + Z⊗Z")

# Step 2: Initial state
psi0 = np.array([1, 0, 0, 0], dtype=complex)  # |00■

# Step 3: Time evolution
t = 1.0
psi_t = runge_kutta_4(psi0, H, t, dt=0.001)

# Step 4: Compute density matrix
rho_full = state_to_density_matrix(psi_t)

# Step 5: Partial trace (reduced state of qubit 0)
rho_A = partial_trace(rho_full, dims=[2, 2], trace_out=1)

# Step 6: Analyze entanglement
P = purity(rho_A)
S = von_neumann_entropy(rho_A)

print(f"State at t={t}: {psi_t}")
print(f"Reduced density matrix:\n{rho_A}")
print(f"Purity: {P:.4f}")
print(f"Entropy: {S:.4f} bits")

# Interpretation:
# - Purity < 1 indicates entanglement
# - Entropy > 0 quantifies entanglement
```

6. Common Use Cases

6.1 Detecting Entanglement

Check if two qubits are entangled by computing the purity of the reduced state:

```
# Entangled state
bell = np.array([1, 0, 0, 1]) / np.sqrt(2)
rho = state_to_density_matrix(bell)
rho_A = partial_trace(rho, dims=[2, 2], trace_out=1)
print(f"Purity: {purity(rho_A)}")  # 0.5 → entangled!

# Product state (not entangled)
product = np.array([1, 1, 0, 0]) / np.sqrt(2)
rho = state_to_density_matrix(product)
rho_A = partial_trace(rho, dims=[2, 2], trace_out=1)
print(f"Purity: {purity(rho_A)}")  # 1.0 → not entangled
```

6.2 Time Evolution Analysis

Track how entanglement evolves over time:

```
H = hamiltonian_from_string("X⊗X + Z⊗Z")
psi0 = np.array([1, 0, 0, 0], dtype=complex)

times = np.linspace(0, 2, 50)
entropies = []

for t in times:
    psi_t = runge_kutta_4(psi0, H, t, dt=0.001)
    rho = state_to_density_matrix(psi_t)
    rho_A = partial_trace(rho, dims=[2, 2], trace_out=1)
    S = von_neumann_entropy(rho_A)
    entropies.append(S)

# Plot entropies vs time to see entanglement dynamics
```

7. Quick Reference

Task	Code
Build H (string)	`hamiltonian_from_string("X⊗Z + Z⊗X")`
Build H (class)	`builder.add_term(['X','Z'], coef=1.0)`
Time evolution	`runge_kutta_4(psi0, H, t, dt=0.001)`
Density matrix	`state_to_density_matrix(psi)`
Partial trace	`partial_trace(rho, dims=[2,2], trace_out=1)`
Purity	`purity(rho)`
Entropy	`von_neumann_entropy(rho)`

Important Notes:

• All Hamiltonians must be Hermitian: H = H†
• State vectors must be normalized: ■ψ|ψ■ = 1
• For n qubits, Hamiltonian is 2■ × 2■ matrix
• Smaller dt gives better accuracy but slower computation
• Energy is conserved: ■ψ(t)|H|ψ(t)■ = constant
• Purity = 1 for pure states, < 1 for mixed states
• Entropy = 0 for pure states, > 0 indicates entanglement

Files Included:

• hamiltonian_builder.py - Hamiltonian construction module
• runge_kutta_hamiltonian.py - Time evolution module
• density_matrix_ops.py - Density matrix operations module
• combined_example.py - Integration examples
• hamiltonian_tutorial.py - Interactive tutorial