QuNet: Python library for studying open qubit systems

Documentation and User Guide

Version 1.2

Unnati Akhouri
`uja5020@psu.edu`
GitHub repository: `https://github.com/UnnatiAkhouri/QuNet`

Abstract

This documentation and user guide is being created as part of a quantum information bootcamp at the Pennsylvania State University. The four week bootcamp will cover how-to download, run, and analyze qubit networks with varying connectivity. The documentation will be updated routinely on Friday of every week during June 2025.

Contents

1 Introduction to Code **3**
1.1 Overview . . . 3
1.2 Key Features . . . 3

2 Motivation **3**
2.1 Scientific Background . . . 4
2.2 Research Applications . . . 4

3 Setup **4**
3.1 Prerequisites . . . 4
3.2 Installation . . . 4
3.2.1 From Source . . . 5
3.3 Verification . . . 5

4 Creating Virtual Environment **5**
4.1 Known Issues . . . 6

5 GitHub Files **6**
5.1 Repository Structure . . . 6
5.2 Key Files Description . . . 6
5.2.1 Core Module Files . . . 6
5.2.2 Utility Files . . . 14
5.2.3 Documentation and Examples . . . 14
5.3 Contributing Guidelines . . . 14

6 Analysis **15**
6.1 Theory . . . 15
6.1.1 Mathematical Framework . . . 15
6.1.2 Numerical Methods . . . 15
6.1.3 Multi-Qubit Systems . . . 15
6.2 Notebooks . . . 15
6.2.1 Tutorial Notebooks . . . 15
6.2.2 Example Notebooks . . . 16

7 Conclusion **16**

1 Introduction to Code

QuNet (previously Qnibble) is a comprehensive Python framework for simulating and analyzing evolution of qubit networks with varying interaction connectivity. This documentation provides a complete guide to understanding, installing, and utilizing the codebase for quantum system simulations.

Throughout the document we will reference papers that facilitate understanding of the various concepts introduced.

1.1 Overview

This package can be used for simulating discrete-in-time evolution of qubit network or circuit with a given or time-varying interaction connectivity. By interaction connectivity which qubits are coupled via interaction terms in the Hamiltonian at a given circuit layer or timestep ℓ. The code was developed to understand the subsystem statistics at the level of one, two and three qubit subsystems within the total system evolving with a Hamiltonian with global symmetry. To that end, the code has particular focus on:

- Qubit Hamiltonians with global conserved quantities
- Qubit networks with varying topology–cyclic chain, fully connected etc.
- Tools to analyze
  - Local, single-qubit dynamical maps
  - Correlation measures like mutual information, concurrence
  - Principal component analysis of local quantities
  - Network measures on mutual information networks like clustering and disparity
  - Thermodynamic quantities like relative entropy and extractable work.
- Visualization of quantum trajectories

1.2 Key Features

- Efficient simulation of qubit networks with modular topology
- Support for custom Hamiltonian definitions
- Built-in visualization tools
- Extensible toolkit for analyzing local subsystem statistics
- Ability to run on cluster and HPC
- Open source

2 Motivation

Can a closed quantum system, evolving under global conservation laws, give rise to subsystems that exhibit persistent out-of-equilibrium dynamics? If so, what is the role of the conserved quantity and do all Hamiltonians within a class of Hamiltonians obeying a symmetry exhibit this or does connectivity have a role to play?

To investigate these questions, we developed a code to study systems can evolve to display different classes of subsystem dynamics over extended periods, a behavior clearly distinguishable from that of approximately thermalizing networks of comparable size. Such quantum systems are fundamental for understanding the emergence of complex and out of equilibrium behavior, with potential connections to biological and condensed matter systems.

In a recent work, we used this code to introduce a novel class of constrained dynamics wherein the quantum system optimizes thermodynamic or quantum information-theoretic measures to explore non-equilibrium state space. The systems studied retain local memory of their initial conditions.

The code allows us to characterize the subsystems by bringing in tools from quantum information—namely phase-covariant dynamical maps—and the theory of open quantum systems. In this way, the subsystems can be studied as an ensemble of open systems, specifically a collection of qubits evolving with phase-covariant dynamics.

Although constraints imposed by the conservation law and the global unitary dynamics of the network bound the distribution of single-qubit dynamics within the ensemble, distinct steady states remain distinguishable by multiple metrics, offering insights into the fundamental behavior of quantum systems far from equilibrium.

By studying the local maps as well as the correlation structure on the qubit network we can draw conclusions about the role correlations play in the long-term local evolution.

Our hope is that this code provides a computational framework for investigating emergent phenomena in quantum statistical mechanics and allows us to examine how constraints and conservation laws can give rise to persistent non-equilibrium states and novel dynamics.

2.1 Scientific Background

The study of qubit system evolution is fundamental to understanding quantum computing, quantum information processing, and quantum many-body physics. Existing simulation tools, often, either are oriented to answer questions ab

2.2 Research Applications

This framework enables researchers to:

- Investigate quantum coherence in multi-qubit systems
- Study the effects of environmental decoherence
- Optimize quantum control protocols
- Validate theoretical predictions with numerical simulations

3 Setup

3.1 Prerequisites

Before installing QuNet, ensure you have the following requirements:

3.2 Installation

We recommend that you use Github desktop for interfacing with the code, especially if you intend to make changes/modify the code. Additionally having a python editor platform like PyCharm will be beneficial if you are to read and edit the code.

Package	**Version**
collections-extended	2.0.2
fqdn	1.5.1
isoduration	20.11.0
jsonpointer	3.0.0
jupyter	1.1.1
mpl-toolkits.clifford	0.0.2
networkx	3.4.2
pip	23.2.1
plotly	6.1.2
scikit-learn	1.6.1
seaborn	0.13.2
statsmodels	0.14.4
tinycss2	1.4.0
tqdm	4.67.1
uri-template	1.3.0
webcolors	24.11.1
wheel	0.41.2

Table 1: Installed Python Packages

3.2.1 From Source

```
git clone https://github.com/UnnatiAkhouri/QuNet.git
cd qunet
pip install -e .
```

3.3 Verification

To verify your installation:

```
import qunet
print(qunet.__version__)
qunet.test()  # Run basic tests
```

4 Creating Virtual Environment

Upon opening the github repository in Pycharm or whichever platform you use, navigate the path to create a virtual environment if not already prompted by the application. In Pycharm, you can follow the following steps to ensure you create a proper virtual environment. Creating a virtual environment ensures that the packages and the version of the packages, including python itself, needed for this library do not interfere with the python and packages on your base. Here are the steps:

1. Go to PyCharm → Settings
2. In the pop-up widow click the project denoted by the project name
3. Click on python interpreter → Add interpreter
4. Click Add local interpreter
5. In the dialog that opens:

- Select Virtualenv Environment from the left panel
- Choose New environment.
- Select a location for your virtual environment (usually in your project folder)
- Choose the base Python interpreter i.e. verison of python
- Click Ok/apply

4.1 Known Issues

- If Jupyter plugin has not been downloaded on pycharm, you will not be able to run the notebooks in pycharm
- If packages have not been downloaded properly, many code functions fail to run. To check which libraries you have, in the terminal type **pip list –not-required** This will give you a list of all the packages you have in the venv.
- matplotlib does not appear as a separate package in the list because it is a dependency of seaborn and plotly.

5 GitHub Files

5.1 Repository Structure

The QuNet repository is organized as follows:

```
qunet/
          README.md
          LICENSE
          Scripts
          Singularity
          src
          data
          Tests
                test_comprehensive.py
                test_density_matrix.py
                test_ket.py
                test_measurements.py
          notebooks/
              many analysis jupyter notebooks
              .ipynb
              images get stored here
```

5.2 Key Files Description

5.2.1 Core Module Files

- **src**: this contains python files that are needed to run the simulations. This module is called in most analysis notebooks as its contains the framework for generating:
  - **ket class**: python file that generates binary strings into a ket belonging to a qubit system of appropriate size in the canonical basis i.e. the binary counting basis eg $|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle$, and $|111\rangle$. Also reorders the basis into *energy-basis* i.e. rearranges rows and columns such that kets with equal number of up state are consecutive eg $|000\rangle, |001\rangle, |010\rangle, |100\rangle, |011\rangle, |101\rangle, |110\rangle$, and $|111\rangle$.

- **DensityMatrix class:** Generates Density matrices out of matrices. Performs usual tasks like basis rotations according to the allowed rotations in the ket class. Has precoded checks for doing matrix operations on matirces with correct basis and number of qubits. Has options to work with sparse or csr matrices. Also, can generate a uncorrelated n-thermal-qubit state if given a string of numbers.
  - **__init__(matrix, basis)**: Initializes the density matrix with a given matrix and basis.
  - **__add__, __mul__, __rmul__, __pow__, __neg__**: Operator overloads for addition, scalar multiplication, matrix multiplication, powers, and negation.
  - **tensor(*others, resultant_basis=None)**: Returns the tensor product with other density matrices.
  - **ptrace(qbits, resultant_dm=True)**: Partial trace over specified qubits, returning the reduced density matrix.
  - **ptrace_to_a_single_qbit(remaining_qbit)**: Returns the population of a single qubit after tracing out others.
  - **ptrace_to_2_qubits(remaining_qubits)**: Returns the reduced density matrix for two qubits.
  - **qbit_basis()**: Returns the density matrix reshaped into a multi-qubit array.
  - **change_to_energy_basis()**: Changes the basis to the energy basis.
  - **change_to_canonical_basis()**: Changes the basis to the canonical basis.
  - **relabel_basis(new_order)**: Relabels the basis according to a new qubit order.
  - **plot()**: Visualizes the density matrix as a heatmap.
  - **data, basis, size, H**: Properties for accessing the matrix data, basis, size, and Hermitian conjugate.
  - **tensor(DMS)**: Utility function to tensor a list of density matrices.
  - **Identity(basis)**: Returns the identity density matrix for a given basis.
  - **qbit(pop)**: Returns a single-qubit density matrix with specified population.
  - **n_thermal_qbits(pops)**: Returns a density matrix for n thermal qubits with given populations.
  - **dm_exp(dm)**, **dm_log(dm)**: Matrix exponential and logarithm of a density matrix.
  - **dm_trace(dm)**: Returns the trace of a density matrix.
  - **permute_sparse_matrix(M, new_order)**, **permute_sparse_matrix_new(m, new_order)**: Utilities for permuting rows/columns of sparse matrices.
  - **conserves_energy(dm)**: Checks if the density matrix conserves energy (nonzero elements only between states of equal energy).
- **Circuit Architechture**: Generates circuit diagrams given a set of gates and measurements sites. Gates are colored by size and labeled by the name given in string. Can also develop random circuit figures with measurements performed at a given probability rates. Can generate the seminal Measurement-Induced-phase-Transition plot as a function of measurement probability.
  - **draw_quantum_circuit(num_qubit, timesteps, gate_sequence)**: Visualizes a quantum circuit diagram for a given number of qubits, timesteps, and a gate sequence.
  - **Shors_algorithm()**: Draws a circuit diagram for Shor's algorithm using the visualization function.

  - **Grovers_algorithm()**: Draws a circuit diagram for Grover's algorithm using the visualization function.
  - **Teleportation()**: Draws a circuit diagram for the quantum teleportation protocol.
  - **random_two_qubit_gate_sequence(num_qubit, timesteps, gate_name)**: Generates a random two-qubit gate sequence for a specified number of qubits and timesteps.
  - **random_two_qubit_gate_sequence_with_measure(num_qubit, timesteps, gate_name, p)**: Generates a random two-qubit gate sequence with measurement gates inserted with probability p.
  - **find_min_cut_trajectory_between_with_measure_robust(a_circuit_sequence, start_index_1, start_index_2, num_trajectories)**: Finds the minimum cut trajectory between two qubits in a circuit sequence, robust to measurement gates.
  - **plot_minimal_cut_system_size_vs_measurement_probability(p_values, qubit_sizes, num_trials)**: Plots the average minimum cuts as a function of system size and measurement probability.
  - **plot_minimal_cut_subsystem_size_separate()**: Plots minimal cuts versus position for different system sizes and measurement probabilities.

- **channels** returns a list of kraus operators that can be applied with given parameter to the qubit chain at the ends. Can be used to study quantum network evolution under noise with a given probability and noise parameters. Krause operators and noise parameters are fully configurable
  - **independent_weak_measurement_kraus(alpha1, alpha2, observable1, observable2)**: Returns Kraus operators for independent weak measurements on two qubits.
  - **correlated_weak_measurement_kraus(alpha, correlation_type)**: Returns Kraus operators for correlated two-qubit weak measurements.
  - **phase_covariant_channel_affine(l1, l3, tau3)**: Constructs a phase covariant channel as an affine map.
  - **phase_covariant_kraus_operators(l3, t3, l1)**: Returns the four Kraus operators for a phase covariant channel.
  - **create_uncorrelated_2qubit_kraus(single_qubit_kraus)**: Forms a two-qubit channel as a tensor product of single-qubit Kraus operators.
  - **create_perfectly_correlated_2qubit_kraus(single_qubit_kraus)**: Forms a two-qubit channel where both qubits experience the same Kraus operator.
  - **two_qubit_depolarizing_kraus(p)**: Returns Kraus operators for a two-qubit depolarizing channel.
  - **independent_dephasing_kraus(p1, p2)**: Returns Kraus operators for independent dephasing on two qubits.
  - **independent_bitflip_kraus(p1, p2)**: Returns Kraus operators for independent bit-flip noise on two qubits.
  - **independent_amplitude_damping_kraus(gamma1, gamma2)**: Returns Kraus operators for independent amplitude damping on two qubits.
  - **correlated_dephasing_kraus(p)**: Returns Kraus operators for correlated dephasing.
  - **pauli_twirling_kraus()**: Returns Kraus operators for a Pauli twirling channel.
  - **CNOT_kraus()**: Returns the CNOT gate as a Kraus operator.

    * **verify_completeness(kraus_ops)**: Checks if a set of Kraus operators satisfies the completeness relation.
    * **cz_gate(), cr_gate(theta), pswap_gate(theta)**: Returns matrices for CZ, controlled rotation, and partial SWAP gates.
    * **embed_edge_channel_full(total_qubits, sub_channel, edge_position)**: Embeds a two-qubit channel into a larger system at a specified edge.
    * **apply_composite_edge_channel(rho, composite_channel, total_qubits)**: Applies a composite channel to a density matrix.
    * **pauli_channel(p_x, p_y, p_z)**, **bit_flip_channel(p)**, **phase_flip_channel(p)**, **depolarizing_channel(p)**, **amplitude_damping_channel(gamma)**, **phase_damping_** **generalized_amplitude_damping_channel(gamma, p)**: Construct various single- and two-qubit noise channels as affine maps.
    * **x_rotation_error_channel(delta)**, **y_rotation_error_channel(delta)**, **z_rotation_error_channel(delta)**: Returns channels modeling rotation errors.
    * **apply_channel(r_aug, channel_matrix)**: Applies a channel matrix to an augmented Bloch vector.
    * **compose_channels(channel1, channel2)**: Composes two channel matrices.
  - **gates**: returns single and two qubit gates if you want to use popular gates instead of energy preserving unitary evolution in your circuit
  - **measurement**: code for all the measurements
    * **temp(qbit)**: Returns the temperature of a single qubit density matrix.
    * **pop(qbit)**: Returns the population (excited state probability) of a single qubit.
    * **pops(dm)**: Returns a list of populations for all qubits in a density matrix.
    * **temps(dm)**: Returns a list of temperatures for all qubits in a density matrix.
    * **average_temp(dm)**: Returns the average temperature across all qubits.
    * **temp_from_pop(pop)**: Computes the temperature from a given population.
    * **pop_from_temp(T)**: Computes the population from a given temperature.
    * **D(dm1, dm2)**: Computes the quantum relative entropy between two density matrices.
    * **D_single_qbits(pop_1, pop_2)**: Computes the relative entropy between two single-qubit populations.
    * **extractable_work(T, dm)**: Calculates the extractable work from a density matrix at temperature T.
    * **extractable_work_of_a_single_qbit(T, pop)**: Calculates extractable work for a single qubit with given population and temperature.
    * **extractable_work_of_each_qubit(dm)**: Returns a list of extractable work values for each qubit in a density matrix.
    * **extractable_work_of_each_qubit_from_pops(pops)**: Returns extractable work for each qubit given a list of populations.
    * **change_in_extractable_work(T_initial, dm_initial, T_final, dm_final)**: Computes the change in extractable work between two states.
    * **entropy(dm)**: Computes the von Neumann entropy of a density matrix.
    * **concurrence(dm)**: Computes the concurrence (entanglement measure) for a two-qubit density matrix.
    * **uncorrelated_thermal_concurrence(dm)**: Computes the concurrence for an uncorrelated thermal two-qubit state.

  - **mutual_information_with_environment(dm, sub_system_qbits)**: Computes the mutual information between a subsystem and its environment.
  - **mutual_information(dm, sub_system_qbits_a, sub_system_qbits_b)**: Computes the mutual information between two subsystems.
  - **mutual_information_of_every_pair(dm)**: Returns the mutual information for every pair of qubits.
  - **two_qbit_dm_of_every_pair(dm)**: Returns the two-qubit reduced density matrix for every pair of qubits.
  - **three_qbit_dm_of_every_triplet(dm)**: Returns the three-qubit reduced density matrix for every triplet of qubits.
  - **four_qbit_dm_of_every_quartet(dm)**, **five_qbit_dm_of_every_quintet(dm)**, **six_qbit_dm_of_every_sextet(dm)**, **seven_qbit_dm_of_every_seventet(dm)**: Return reduced density matrices for all quartets, quintets, sextets, and septets of qubits, respectively.
  - **relative_entropy_of_every_pair(dm)**: Computes the relative entropy for every pair of qubits.
  - **monogamy_of_mutual_information(dm, a, b, c)**: Computes the monogamy of mutual information for three subsystems.
  - **subaddativity(dm, a, b)**: Computes the subadditivity of entropy for two subsystems.
  - **strong_subaddativity(dm, a, b, c)**: Computes the strong subadditivity of entropy for three subsystems.
  - **mutual_information_of_every_pair_dict(dm_dict)**: Computes mutual information for every pair in a dictionary of density matrices.
  - **concurrence_of_every_pair_dict(dm_dict)**: Computes concurrence for every pair in a dictionary of density matrices.
  - **strong_subaddativity_of_every_triplet_dict(dm_dict)**: Computes strong subadditivity for every triplet in a dictionary of density matrices.
  - **monogamy_of_mutual_information_of_every_triplet_dict(dm_dict)**: Computes monogamy of mutual information for every triplet in a dictionary of density matrices.

- **Order rules**: code that describes the cost fucntion used by the quantum network to optimize a cost and use a particular circuit sequence
  - **random(...)**: Selects a random order of qubit groupings based on the specified connectivity.
  - **greedy(...)**: Chooses the order that maximizes the sum of deviations of updated populations from the average, favoring the most "out-of-equilibrium" configuration.
  - **therm(...)**: Selects the order that minimizes the sum of deviations of updated populations from a fixed thermal value (e.g., 0.225).
  - **strongest_maximizes(...)**: Picks the order that maximizes the increase in extractable work for the qubit with the largest change in extractable work.
  - **landscape_maximizes(...)**: Chooses the order that maximizes the total change in extractable work across all qubits.
  - **mimic(...)**: Constructs an order by pairing qubits to mimic the population changes of their neighbors, based on previous steps and connectivity.
  - **thermodynamic(...)**: Selects the order that maximizes the increase in extractable work for the qubit with the smallest change in extractable work, using a fixed thermal population as reference.

- **orders**: precoded list of all k2 local pairs possible for a given connectivity
  - **all_c2_2local_orders(num_qbits, chunk_size)**: Returns all possible 2-local groupings for a chain of qubits with 2-local connectivity.
  - **all_c4_2local_orders(num_qbits, chunk_size)**: Returns all possible 2-local groupings for a 4-qubit ring.
  - **all_c5_orders(num_qbits, chunk_size)**: Returns all possible groupings for a 5-qubit ring.
  - **all_c5_2local_orders(num_qbits, chunk_size)**: Returns all possible 2-local groupings for a 5-qubit ring.
  - **all_c6_orders(num_qbits, chunk_size)**: Returns all possible groupings for a 6-qubit ring.
  - **all_c6_2local_orders(num_qbits, chunk_size)**: Returns all possible 2-local groupings for a 6-qubit ring.
  - **all_c7_orders(num_qbits, chunk_size)**: Returns all possible groupings for a 7-qubit ring.
  - **all_cN_2local_orders(num_qbits, chunk_size)**: Returns all possible 2-local groupings for a ring of arbitrary size.
  - **n_random_c2_2local_orders(num_qbits, chunk_size)**: Returns a random 2-local grouping for a chain of qubits.
  - **n_random_c4_2local_orders(num_qbits, chunk_size)**: Returns a random 2-local grouping for a 4-qubit ring.
  - **n_random_c5_orders(num_qbits, chunk_size, n)**: Returns n random groupings for a 5-qubit ring.
  - **n_random_c5_2local_orders(num_qbits, chunk_size)**: Returns a random 2-local grouping for a 5-qubit ring.
  - **n_random_c6_2local_orders(num_qbits, chunk_size)**: Returns a random 2-local grouping for a 6-qubit ring.
  - **n_random_cN_2local_orders(num_qbits, chunk_size)**: Returns a random 2-local grouping for a ring of arbitrary size.
  - **n_random_c6_orders(num_qbits, chunk_size, n)**: Returns n random groupings for a 6-qubit ring.
  - **n_random_c7_orders(num_qbits, chunk_size, n)**: Returns n random groupings for a 7-qubit ring.
- **disordered networks**:This module provides functions for generating and analyzing disordered quantum networks, including random connectivity, disorder realizations, and related utilities.
  - **generate_random_connectivity(num_qbits, p)**: Generates a random connectivity graph for a given number of qubits, where each possible edge is included with probability p.
  - **generate_disorder_realization(num_qbits, disorder_strength)**: Produces a single realization of disorder (e.g., random on-site energies) for the network.
  - **generate_multiple_disorder_realizations(num_qbits, disorder_strength, n)**: Generates n independent disorder realizations for statistical analysis.
  - **apply_disorder_to_hamiltonian(hamiltonian, disorder)**: Modifies a given Hamiltonian by adding a disorder realization to its diagonal elements.
  - **random_network_orders(num_qbits, chunk_size, connectivity_matrix)**: Returns random groupings (orders) of qubits based on a provided connectivity matrix.

    * **get_connectivity_matrix(num_qbits, edges)**: Constructs a connectivity matrix from a list of edges for a network of n qubits.
    * **analyze_network_properties(connectivity_matrix)**: Computes and returns properties of the network, such as degree distribution or connected components.

- **random unitary**: This module provides functions for generating random unitary matrices and applying them to quantum systems. These are useful for simulating generic quantum operations or noise.
    * **random_unitary(dim)**: Generates a random unitary matrix of dimension `dim` using the Haar measure.
    * **apply_random_unitary(dm, qubits)**: Applies a random unitary operation to the specified qubits of a given density matrix `dm`.
    * **random_unitary_ensemble(dim, n)**: Generates an ensemble of n random unitary matrices of dimension `dim`.
- **relabel basis check**: This module provides functions to verify and manipulate the labeling of quantum basis states, ensuring consistency and correctness in basis transformations and relabeling operations.
    * **relabel_basis(dm, new_basis)**: Relabels the basis of the given density matrix `dm` to the specified `new_basis`.
    * **check_basis_consistency(dm, basis)**: Checks if the density matrix `dm` is consistent with the provided `basis` labeling.
    * **get_basis_permutation(old_basis, new_basis)**: Computes the permutation required to transform from `old_basis` to `new_basis`.
    * **apply_basis_permutation(dm, permutation)**: Applies a basis permutation to the density matrix `dm` according to the given `permutation`.
- **setup** This module is typically used to configure the packaging and installation of a Python project. It defines metadata, dependencies, and entry points for the package
    * **setup(...)**: The main function that configures the package, specifying its name, version, author, description, required dependencies, and other metadata for distribution.
- **simulation** This module contains functions and classes to run quantum system simulations, typically orchestrating the evolution of density matrices, applying order rules, and collecting measurement results.
    * **run_simulation(params)**: Runs a full simulation using the provided parameters, including initial state, connectivity, order rule, and number of steps.
    * **apply_order_rule(dm, order_rule, ...)**: Applies a specified order rule to the current density matrix and updates the system state.
    * **collect_measurements(dm, ...)**: Gathers measurement results (e.g., populations, mutual information) from the current density matrix.
    * **SimulationResult**: A class or data structure to store and organize the results of a simulation, such as time series of observables.
- **simulation with channels**: This module contains functions and classes for simulating quantum systems where quantum channels (such as noise, decoherence, or general CPTP maps) are applied during the evolution, in addition to unitary operations. This allows for more realistic modeling of open quantum systems.
    * **run_simulation_with_channels(params)**: Runs a simulation where quantum channels are applied at each step, in addition to unitary evolution, using the provided parameters.

    - **apply_channel(dm, channel, qubits)**: Applies a specified quantum channel to the given qubits of the density matrix `dm`.
    - **collect_channel_measurements(dm, ...)**: Collects measurement results after the application of channels, such as populations, entropies, or fidelities.
    - **SimulationWithChannelsResult**: A class or data structure to store and organize the results of simulations involving channels.
  - **swap gate reorder** this provides functions to check that the swap and reordering of basis is working correctly.

- **Scripts**
  - **hdf5merge**:
    - **merge_groups(group1, group2, group_out)**: Recursively merges datasets and groups from `group1` into `group_out`, avoiding overwriting datasets that exist in both sources.
    - **merge_hdf5_files(directory)**: Finds all `.hdf5` files in the specified directory and merges their contents into a single output file, using `merge_groups` for the merging logic.
  - **simulation CLI**: the code we use to work the code with the cc-star cluster.
    - **execute(file_name, connectivity, order_rule_name, unitary_energy_subspace, unitary_seed, num_steps, initial_pops, evolution_generator_type, chunk_size, channel_name, channel_prob, verbosity, first_10_order, return_all_dms)**: Orchestrates the setup and execution of a quantum simulation, including system initialization, unitary/channel construction, running the simulation, and saving results. Handles both unitary and noisy (channel) evolutions.
    - **save_data(file_name, data, connectivity, unitary_energy_subspace, unitary_seed, order_rule_name, measurment, num_qubits)**: Saves simulation results (populations, density matrices, orderings) to an HDF5 file, organizing data by simulation parameters.
    - **main block**: Parses command-line arguments using `argparse`, prepares simulation parameters, and calls `execute()` to run the simulation.
  - **singularity test**
    - **Imports and setup**: Adds parent directory to `sys.path`, imports required modules from `src`.
    - **Argument parsing**: Reads command-line arguments for system size (`N`), order type, and index.
    - **Initial parameters**: Sets chunk size, number of chunks, identity operator, number of iterations, and initial populations.
    - **Orderings**: Generates different types of qubit orderings (`gas`, `c5`, `c6`, `c7`, `messenger`) using functions from `orders`.
    - **Unitary construction**: Builds a block-diagonal unitary operator from subspace unitaries and identity operators.
    - **System initialization**: Creates an initial thermal state density matrix and changes its basis.
    - **Simulation**: Runs the simulation with the specified measurements, orderings, and unitary, collecting results.
    - **Data saving**: Saves the populations and extractable work results using `sim.save_data`.
  - **small subsystem energy subspace**

    * **accessible_states(state, partition_size)**: Returns all possible permutations of a given state, used to explore accessible configurations within a partition.
    * **partition(lst, n, offset=0)**: Splits a list (state) into partitions of size n, with an optional offset for rolling the list.
    * **all_permutations(partition)**: Computes all possible ordered combinations of permutations for each partition, returning the set of all such combinations.
    * **ordered_combinations(l1, l2)**: Returns all ordered concatenations of elements from two lists.
    * **do(num_qbits, partition_size)**: Constructs a matrix representing which states can be entangled via permutations within partitions, for a system of given size and partitioning.
    * **Main execution**: Builds the entanglement matrix for 8 qubits with partition size 2, constructs a density matrix, changes its basis, and plots it.
  - **testing**: a basic testing script
    * **Imports and setup**: Adds parent directory to `sys.path`, imports modules for measurements, density matrix, simulation, orders, and random unitary generation.
    * **Simulation execution**: Calls `cleo.execute()` with specified parameters to run a quantum simulation and stores the result in `data`.
    * **Output**: Prints the first element of `data` and computes the mutual information of every pair using `measure.mutual_information_of_every_pair_dict()`.

- **QuNET bootcamp 2025**

5.2.2 Utility Files

-
-

5.2.3 Documentation and Examples

- `notebooks/`: Jupyter notebooks with tutorials and examples
- `docs/`:
- `examples/`: Python script examples

5.3 Contributing Guidelines

To contribute to QuNet:

1. Fork the repository
2. Create a feature branch: `git checkout -b feature-name`
3. Make your changes and add tests
4. Run the test suite: `pytest`
5. Submit a pull request

6 Analysis

6.1 Theory

6.1.1 Mathematical Framework

6.1.2 Numerical Methods

6.1.3 Multi-Qubit Systems

For N-qubit systems, the Hilbert space dimension scales as 2^N. QuNet handles this through:

- Efficient sparse matrix representations
- Tensor product operations
- Memory-optimized state vector storage

6.2 Notebooks

The QuNet package includes comprehensive Jupyter notebooks for learning and experimentation:

The analysis notebooks assume that the data is stored in the hdf5 file format. This will be the case if you choose to use this code to develop your data via local-on-machine simulation or on-cluster simulation. However, if you want to use the code to analyze external data, not in the hdf5 format, you will have to modify the functions discussed in this section.

For more information on the hdf5 file format read this. `https://www.earthdata.nasa.gov/about/esdis/esco/standards-practices/hdf5`

6.2.1 Tutorial Notebooks

Basic Tutorial (`tutorial_basic.ipynb`) This notebook covers:

- Setting up single and multi-qubit systems
- Defining custom Hamiltonians
- Basic time evolution
- Visualization of quantum trajectories
- Computing expectation values

Example workflow:

```python
import qunet as qn
import numpy as np

# Create a two-qubit system
system = qn.QubitSystem(n_qubits=2)

# Define initial state |01
psi0 = qn.basis_state([0, 1])

# Create Hamiltonian (e.g., Ising model)
H = qn.Hamiltonian.ising(coupling=0.1, field=0.05)

# Time evolution
times = np.linspace(0, 10, 100)
evolution = qn.evolve(psi0, H, times)

```

```
# Visualize results
qn.plot_populations(evolution, times)
```

Advanced Tutorial (`tutorial_advanced.ipynb`) Advanced topics include:

- Time-dependent Hamiltonians
- Tz shift parameter
- Concurrence
- MI networks
- MMI

6.2.2 Example Notebooks

Physics Applications

-
-

Benchmarking and Validation

-

7 Conclusion

QuNet provides a robust, well-documented framework for quantum system simulation. The combination of efficient numerical methods, comprehensive documentation, and interactive tutorials makes it suitable for both educational use and research applications.

For the latest updates, bug reports, and feature requests, please visit the GitHub repository at `https://github.com/UnnatiAkhouri/QuNet`.

References