A matrix-product-operators simulator of Lindblad dynamics

Haggai Landa,[1, *] [*and...*],[1] Gal Shmulovitz,[2] Eldor Fadida,[2] and Grégoire Misguich[3, 4, †]

[1]*IBM Quantum, IBM Research Haifa, Haifa University Campus, Mount Carmel, Haifa 31905, Israel*
[2]*School of Electrical Engineering, Faculty of Engineering, Tel Aviv University, Tel Aviv 69978, Israel*
[3]*Institut de Physique Théorique, Université Paris-Saclay, CEA, CNRS, 91191 Gif-sur-Yvette, France*
[4]*Laboratoire de Physique Théorique et Modélisation, CNRS UMR 8089, Université de Cergy-Pontoise, 95302 Cergy-Pontoise, France*

I. INTRODUCTION

A. General background

The state of an open quantum system is defined by a density matrix ρ. When the system is coupled to a memory-less bath, the time evolution is often described using the master equation

$$\frac{\partial}{\partial t}\rho = \mathcal{L}[\rho] \equiv -\frac{i}{\hbar}[H,\rho] + \mathcal{D}[\rho], \tag{1}$$

where $[\cdot,\cdot]$ is the commutator of two operators, and the notation $\mathcal{L}[\rho]$ implies that the Liouvillian $\mathcal{L}$ is a (linear) superoperator that maps the operator ρ to another operator (in this case, the time derivative of ρ). The unitary evolution due to interactions and coherent driving terms is generated by the (possibly time-dependent) Hamiltonian H, while $\mathcal{D}[\rho]$ is a superoperator (sometimes known as the dissipator, or Lindbladian) that accounts for incoherent dephasing and relaxation processes due to the environment.

B. Simulator scope

We introduce a numerical simulator of noisy density-matrix dynamics, specifically Lindbladian dynamics. The solver is based on matrix product states and matrix product operators for the representation of the many-qubit density matrix and its evolution. The solver is written in C++ and uses internally the open source library iTensor []. The solver integrates the dynamics in fixed time steps τ up to the final time t_f, using a Trotter expansion of order up to the fourth.

The simulator supports dynamics of qubits described as two-level systems, in a uniform rotating frame (i.e. with fixed coefficients), with arbitrary on-site Hamiltonian parameters, two-qubit interactions [flip-flop (XY) and Ising] on an arbitrary connectivity graph, and three dissipator jump operators $(\sigma^+, \sigma^-, \sigma^z)$. It is estimated that a few tens of qubits (20-30) in a 'heavy-hex' lattice configuration (as in IBM Quantum Falcon devices) could be simulated in the rotating frame up to times corresponding to the steady state with realistic parameters, on a strong multi-core node. The exact feasible number of qubits depends both on the driving parameters, the dissipation, and the topology. Earlier 1D and 2D simulations using this solver are reported in [1–3].

In the next version, we will introduce general d-level qudit dynamics (suitable for transmon qubits for example), with more general Hamiltonian and Lindbladian parameters, and time-dependence in the parameters (allowing to directly integrate time-dependent drives in the lab frame).

In Sec. II the supported model and its initial state and observables are described. In App. A the input and output parameters of the Python interface for the simulator wrapper are described. In App. B the executable input and output parameters and file formats are specified.

II. MODEL

A. Lindbladian

We model N interacting qubits in a planar lattice with an arbitrary connectivity. The qubits are two-level systems, and we use the Pauli matrices at each site (with the sites indexed by i),

$$\sigma_i^a, \qquad a = \{x, y, z\}, \tag{2}$$

and the ladder operators

$$\sigma_i^{\pm} = \sigma_i^x \pm i\sigma_i^y/2. \tag{3}$$

The Hamiltonian is defined in the rotating frame with respect to the frequency of an identical drive applied to all qubits (which are not necessarily identical themselves, however, and may be driven at different amplitudes). All parameters defined below are assumed constant in this frame. The details of the lab frame and the transformation are given in App. C. Decomposing the Hamiltonian in the rotating frame into the sum of on-site terms and the interaction part T, we have

$$H/\hbar = \sum_i \frac{1}{2}\left[h_{z,i}\sigma_i^z + h_{x,i}\sigma_i^x + h_{y,i}\sigma_i^y\right] + T. \tag{4}$$

[*] haggai.landa@ibm.com
[†] gregoire.misguich@ipht.fr

with the interaction being

$$T = -\sum_{i}^{N}\sum_{j\neq i}^{N}\left(J_{ij}\sigma_i^{+}\sigma_j^{-} + \text{h.c.} + \frac{1}{2}J^z_{ij}\sigma_i^z\sigma_j^z\right) = -\frac{1}{2}\sum_{i}^{N}\sum_{j\neq i}^{N}\left(J_{ij}\sigma_i^x\sigma_j^x + J_{ij}\sigma_i^y\sigma_j^y + J^z_{ij}\sigma_i^z\sigma_j^z\right). \quad (5)$$

We treat three typical dissipator terms in Eq. (1),

$$\mathcal{D} = \sum_j \mathcal{D}_j, \quad (6)$$

with

$$\mathcal{D}_0[\rho] = \sum_i g_{0,i}\left(\sigma_i^{+}\rho\sigma_i^{-} - \frac{1}{2}\{\sigma_i^{-}\sigma_i^{+}, \rho\}\right), \quad (7)$$

$$\mathcal{D}_1[\rho] = \sum_i g_{1,i}\left(\sigma_i^{-}\rho\sigma_i^{+} - \frac{1}{2}\left\{\sigma_i^{+}\sigma_i^{-}, \rho\right\}\right). \quad (8)$$

$$\mathcal{D}_2[\rho] = \sum_i g_{2,i}\left(\sigma_i^z\rho\sigma_i^z - \rho\right), \quad (9)$$

The physical process corresponding to Eqs. (7)-(8) is energy exchange with a thermal bath, and Eq. (9) corresponds to ("pure") dephasing in xy plane.

B. Initial state

The simulator takes as input an initial state that can be a pure Pauli product state,

$$\rho(t=0) = |\psi_0\rangle\langle\psi_0|, \qquad |\psi_0\rangle = \prod_i |\pm a_i\rangle, \quad (10)$$

with $|\pm a_i\rangle$ a Pauli eigenstate of qubit i;

$$\sigma_i^a|\pm a_i\rangle = \pm|\pm a_i\rangle. \quad (11)$$

In addition, the simulator can also save its internal state representation to a file, and load a saved state to use it as the initial state.

C. Observables

For the final time t_f and intermediate times t_k, the output of the simulator consists of

1. Time-dependent single-qubit expectation values,

$$\langle\sigma_i^a(t_k)\rangle. \quad (12)$$

2. Equal-time two-qubit correlators,

$$\left\langle\sigma_i^a(t_k)\sigma_j^b(t_k)\right\rangle. \quad (13)$$

These are also referred to as Pauli expectation values (of one and two qubits, respectively), at time t_k.

[1] K. Bidzhiev and G. Misguich, Phys. Rev. B **96**, 195117 (2017).

[2] H. Landa, M. Schiró, and G. Misguich, Physical review letters **124**, 043601 (2020).

[3] H. Landa, M. Schiró, and G. Misguich, Physical Review B **102**, 064301 (2020).

Appendix A: Python Interface

1. Input parameters

The solver supports the following parameters as input in the Python interface. If there is a default value, it is given with an equality sign, otherwise it must be specified and an exception is thrown if it is not passed.

Basic parameters

1. `N`. The number of qubits in the lattice.
2. `t_final`. The final (total) simulation time, t_f.
3. `tau`. The discrete time step τ for the time evolution.
4. `input_file = ‘’`. The file name to generate with the input parameters for the executable. If any empty string is given, a temporary system file is allocated and used.

Hamiltonian coefficients

1. `h_x = 0`. The $h_{x,i}$ coefficient in Eq. (4). If a vector is given, it specifies $h_{x,i}$ for each qubit. If a scalar is given, it is uniform for all qubits.

2. `h_y = 0`. The $h_{y,i}$ coefficient in Eq. (4). The syntax and usage are identical to that of `h_x`.

3. `h_z = 0`. The $h_{z,i}$ coefficient in Eq. (4). The syntax and usage are identical to that of `h_x`.

4. `J_z = 0`. The J^z_{ij} coefficient in Eq. (5). If a matrix is given, it specifies J^z_{ij} for each pair of qubits. If a scalar is given, it is uniform for all qubits of a lattice, as specified below.

5. `J = 0`. The J_{ij} coefficient in Eq. (5). The syntax and usage are identical to that of `J_z`. If either `J` or `J_z` are a matrix, then the other one must be either a matrix as well, or `0`.

Dissipation coefficients

1. `g_0 = 0`. The $g_{0,i}$ coefficient in Eq. (7). The syntax and usage are identical to that of `h_x`.

2. `g_1 = 1`. The $g_{1,i}$ coefficient in Eq. (8).

3. `g_2 = 0`. The $g_{2,i}$ coefficient in Eq. (9).

Initial state

1. `init_Pauli_state = '+z'`. Either a length-N vector of two-character strings of the form $\pm a$, or a single such string. Each string indicates the initial state of qubit i is a Pauli state as detailed in Sec. II B, and a single string indicates an identical initial state for all qubits.

2. `load_state_file = ''`. A file name as previously saved using the simulator, to load the initial state from. If this string is nonempty, then `init_Pauli_state` must be an empty string.

Lattice specification

If both parameters `J` and `J_z` specifying the qubit coupling are scalar (and not a matrix), then the simulator generates a lattice coupling that is specified using the following parameters.

1. `l_x = 0`. The length of the lattice along the x dimension (l_x). If left at the default 0 value, then the number qubits `N` is used, and `l_y` must take its default value (1) as well.

2. `l_y = 1`. The length of the lattice along the y dimension (l_y).

3. `b_periodic_x = False`. Whether periodic boundary conditions are applied along the x dimension. If True, then `l_y = 1` is required.

4. `b_periodic_y = False`. If True, periodic boundary conditions in the y direction are used.

The supported configurations are a 1D chain or a 2D rectangular strip (both with open boundary conditions), a 1D ring (with periodic boundary conditions), or a 2D cylinder (with periodic boundary conditions along the y direction).

Numerical simulation control

1. `Trotter_order = 4`. Trotter approximation order, Possible values are 2,3,4.

2. `max_dim_rho = 500`. Maximum bond dimension for density matrices.

3. `cut_off_rho = 1e-9`. Maximum truncation error for density matrices.

4. `b_force_rho_trace = True`. Whether the trace of the density matrix be forced to 1 at every time step, irrespective of finite-step errors.

5. `b_force_rho_hermitian = True`. Whether the density matrix be forced to be Hermitian before measuring observables. This is recommended, as it reduces some errors.

Observables and output

1. `save_state_file = ''`. A file name where the simulator writes the final state to. Two files will be saved, whose names will be modified by adding the number of qubits to the file name, and using two different extensions.

2. `output_file = 'out'`. A file name where the simulator writes the observables to. Possibly two files will be saved, using two different extensions `.1q_obs.dat` and `.2q_obs.dat`, with the single-qubit and two-qubit observables as specified below.

3. `output_step = 1`. How often (in units of `tau`) the observables are computed. For 0 no observables will be computed.

4. `1q_indices = []`. A list of integers that specify the qubits for calculating single-qubit observables as in Eq. (12). If left empty it will default to all qubits.

5. `1q_components = ['X', 'Y', 'Z']`. A list of strings that specify the Pauli observables to compute for all qubits given in `1q_indices`.

6. `2q_indices = []`. A list of integer tuples that specify the qubit pairs for calculating single-qubit observables as in Eq. (13). If left empty it will default to all qubit pairs.

7. `2q_components = ['XX', 'YY', 'ZZ']`. A list of strings that specify the two-qubit Pauli observables to compute for all qubits given in `2q_indices`.

2. Call interface

The solver is exposed using the Python class `MPOLindbladSolver`. The class has the following methods.

1. `@staticmethod build_input_file(parameters: Dict)`. Build the input file for the solver based on the parameters dictionary.

2. `@staticmethod load_output(s_output_file: str) -> Dict`. Load the output file(s) using the given file name, and generate the result dictionary.

3. `@staticmethod execute(s_input_file: str)`. Execute (synchronously) the solver, utilizing the input file parameter.

4. `solve(parameters: Dict)`. This methods combines the generation of an input file for the solver, its synchronous execution, and the construction of the result dictionary. The `parameters` and `result` dictionaries are stored in class fields for further processing.

3. Result

The result dictionary stores the output (observables) in the following format;

1. `1q_observables: Dict`. A dictionary for the one-qubit observables with the keys being a tuple with the format:

   `(qubit: int, Pauli: str, time: float)`.

   Each value is the expectation value for the specific key parameters, for example:

   `{(1, ‘X’, 0): 0, (2, ‘Z’, 0.1): 0.8}`.

   In this example the result object contains two elements: the Pauli-X expectation value of qubit 1 at time 0 seconds, and the Pauli-Z expectation value for qubit 2 at time 0.1 seconds.

2. `2q_observables: Dict`. A dictionary for the two-qubit observables with the keys being a tuple with the format:

   `(qubit, qubit, Pauli: str, time: float)`.

   For example:

   `{(1, 2, ‘ZZ’, 0.1): 0.7}`.

   In this example the result object contains the Pauli correlation of qubit 1 on the ‘X’ axis and qubit 2 on ‘Z’ axis at time 0.1 seconds.

Appendix B: Solver Executable Interface

1. Solver input

The solver executable accepts a single parameter in its command line, which is the file name of an input file. In the input file each parameter is specified on a separate line, which contains the parameter name, a space, the equal sign “=”, a space, and a single value or a comma-separated list of values (without spaces).

The parameters that the executable accepts have identical names and meaning as the parameters detailed above for the Python interface, with the following exceptions;

1. `J, J_z`. These two executable parameters are passed as a single scalar value (if the Python parameter is scalar), or as a list (no matrix format is supported). If either `J` or `J_z` is a matrix in the Python, then the list consists of all entries which are nonzero in either one of the two matrices. In this case, there are two additional parameters used in order to describe the qubit pairs to which the values correspond:

2. `A_bond_indices, B_bond_indices`. List of the indices of the first and second qubit (respectively) of each entry in the lists `J, J_z` (that must be of identical length).

2. Solver output files

The solver executable generates up to three files: a saved state (at the simulation end), observables at requested times, and a log file. The format of the save state file is not described here. The log file is textual. The format of the observable files is detailed below.

1. The one-qubit observables file is ordered in a table format, in which the headers are: `Component`, `time_t`, `site_i`, `ExpectationValue`. The `Component` is the Pauli operator $\{X, Y, Z\}$, `time_t` is the time that the observable is taken, `site_i` is the qubit index and `ExpectationValue` is the pauli expectation value as in Eq. (12). This will be filled row by row, for each time step there will be small tables as the number of qubits, each small table has the specified components ordered by qubit index.

2. The two-qubit observables file is ordered in a table format as well, with headers being: `Component`, `time_t`, `site_i`, `site_j`, `ExpectationValue`. The `Component` is the two specified correlators , `time_t` is the time of the observation, `site_i` is the first qubit index, `site_j` is the second qubit index and `ExpectationValue` is the two qubit pauli correlation value as in Eq. (13). This will be filled row

by row, same order as in one qubit file, except each time step table has all possible combinations of qubits requested.

Appendix C: The lab frame and rotating frame

TBD.