January 24, 2025

In this document, we describe different functions that contribute to the computation of two-point functions for the Bose-Hubbard model.

1 Computing the Quantum Speed Limit

1.1 QSL Computation for the Bose-Hubbard Model

This script performs Quantum Speed Limit (QSL) computations for the Bose-Hubbard model, analyzing the evolution of an initial system state and computing bounds on scrambling behavior using out-of-time-ordered correlators (OTOCs).

Dependencies

Parameter Definitions

The following parameters are set for the simulation:

- `N = 3` – Number of bosons in the chain.
- `M = 3` – Number of sites in the chain.
- `J = 4.0` – Hopping parameter.
- `U = 8.5` – On-site interaction potential.
- `T = eltype(J)` – Data type inferred from the hopping parameter.
- `beta = 1.0` – Inverse temperature of the system.
- `t_stop = 0.2` – Maximum time for evolution.
- `num_points = 30` – Number of time points for analysis.

The time vector is generated using:

```
times = np.linspace(0, t_stop, num_points)
```

State Preparation

Step 1: Define Pure System State The system state is initialized with only one boson:

```
pure_system = zeros(T, N+1, N+1)
pure_system[N, N] = 1.0
```

Step 2: Thermal Density Matrix Computation A thermal density matrix at the specified temperature is computed using:

```
thermal_dm = thermal_state(beta, N, M, J, U)
```

This results in a block-diagonal density matrix for the environment.

Step 3: Tensor Product The initial density matrix is constructed by taking the tensor product of the pure system and the thermal bath:

```
result_dm = sparse(kron(pure_system, thermal_dm))
```

Step 4: Limiting the Joint State and Applying Quench The initial density matrix is limited to N bosons in total:

```
init_state = limit_dm(result_dm, N, M)
```

Hamiltonian Definitions

- `H = BoseHubbard(N, M, J, U , :OBC)`
  Defines the full Hamiltonian for N bosons on M sites with open boundary conditions.
- `sys_ham = RBoseHubbard(N, 2, 0.0, U)`
  Hamiltonian for the system with N bosons and 1 site.
- `bath_ham = RBoseHubbard.([N+1,N], M, J, U)`
  Defines the bath Hamiltonian with N bosons on $M - 1$ sites.

Eigenvalues and eigenvectors of the bath Hamiltonian are computed as:

```
eigenvals, eigenvecs = eigen!(Matrix(bath_ham[2].H))
```

Computing QSL Bounds

Lists to Store Results Empty lists are initialized to store computed bounds and entropy values:

```
renyi_ent_list = []
bound_list = []
bound_list_born = []
spec_bound_list = []
spec_bound_list_born = []
```

Time Evolution and Computation For each time step, the density matrix is evolved,the Renyi entropy and the Q is calculated.

```
@showprogress for (i,t) in enumerate(times)
    rho_t = time_evol_state(init_state, H, t )
    rho_int_t = interaction_picture(NBasis(N,M), U, rho_t)

    rho_S = partial_trace_bath(rho_int_t, N, M)
    push!(renyi_ent_list, renyi_entropy(rho_S))

    rho_B = partial_trace_system(rho_int_t, size(init_state,1),N,M)

    qsl = QSL_OTOC(t, J, N, sys_ham, bath_ham, eigenvecs, rho_B)
    qsl_born = QSL_OTOC(t, J, N, sys_ham, bath_ham, eigenvecs, thermal_dm)

    push!(bound_list, real(qsl.state_bound))
    push!(spec_bound_list, real(qsl.spectral_bound))
    push!(bound_list_born, real(qsl_born.state_bound))
    push!(spec_bound_list_born, real(qsl_born.spectral_bound))
end
```

Plotting Results

The computed bounds and entropy values are plotted:

```
plot(
    times,
    [exp.(-2 * real(bound_list)),
     exp.(-2 * real(bound_list_born)),
     exp.(-spec_bound_list),
     exp.(-spec_bound_list_born),
     exp.(-renyi_ent_list)],
    label=["State space" "State space + Born" "Liouv space" "Liouv space + Born" "OTOC"]
)
xlabel!("time")
ylabel!("QSL")
savefig("qsl_bh.pdf")
```

Explanation of Plotted Results

- **State space**: The quantum speed limit bound computed in state space.
- **State space + Born**: Bound considering the Born approximation.
- **Liouv space**: Quantum speed limit calculated in the Liouville space.
- **Liouv space + Born**: Liouville space bound under the Born approximation.
- **OTOC**: The computed out-of-time-ordered correlator value.

Conclusion

This script sets up a complete workflow to compute the QSL for a Bose-Hubbard system, incorporating system-bath interactions and verifying bounds using numerical simulations.

2 Basis Functions

2.1 `OpenQSL/src/basis.jl/NBasis`

This function generates a basis for exactly N particles in M sites using the occupation number representation. This is the basis that is associated with the `BoseHubbardN,M,J,U,bndr` functions which is the unitary Bose Hubbard model on M sites with N particles. It creates an instance that contains the following four attributes:

- `NBasis(N,M).N` – Returns the number of particles in the system.
- `NBasis(N,M).M` – Returns the number of sites.
- `NBasis(N,M).eig_vecs` – Provides the list of basis vectors generated.
- `NBasis(N,M).tags` – Contains tags associated with the eigenvectors obtained through hashing, which helps in sorting these eigenvectors efficiently, allowing binary search operations.

When working with bosons, we frequently switch between the number states basis and the computational basis. For example, consider a Bose-Hubbard chain with 6 bosons and 6 sites. The Hilbert space size in this case is 462. Each number state, such as:

$$|\psi\rangle = |2,1,1,1,1,0\rangle$$

is mapped to a 462-dimensional unit vector $|0,0,0,\ldots,1,0,\ldots\rangle$ in the computational basis.

In the code, to convert from computational to number basis, we use:

```
number_basis_ket = State(comp_basis_ket, BH[2].basis)
```

To perform the reverse transformation:

```
comp_basis_ket = dense(number_basis_ket, BH[2].basis)
```

Here, `BH[2]` refers to the Hilbert space containing N bosons. Similarly, `BH[1]` and `BH[3]` correspond to subspaces with $N+1$ and $N-1$ bosons, respectively.

2.2 `OpenQSL/src/basis.jl/RBasis`

This function is used to create a basis to express the reduced density matrix of the environment of a finite sized Bose-Hubbard chain. If suppose you started out with a finite chain of N particles on M sites and your environment consists of the last M-1 sites. Then you can have anywhere from 0 to N particles on those M-1 sites. The function `RBasis(N,M)` creates a basis that consists of a list of eigenvectors from the `NBasis` function, stacked together for different particle numbers. Specifically, it combines the eigenvectors from `NBasis(i, M-1).eig_vecs` where i runs from N to 0.

For example, `RBasis(2,3).eig_vecs` gives you {

$$|2,0\rangle, |1,1\rangle, |0,2\rangle$$

,

$$|1,0\rangle, |0,1\rangle$$

}

3 Hamiltonian Function

3.1 `OpenQSL/src/hamiltonian.jl/BoseHubbard`

The function:

```
function BoseHubbard(N, M, J, U, bndr)
```

creates an instance of the type `BoseHubbard`, which contains three attributes: `basis`, `lattice`, and `H`.

For example, to extract the Hamiltonian sparse matrix for a certain choice of parameters, we write:

```
N=6
BH = BoseHubbard([N+1, N, N-1], M=6, J=4.0, U=16.0, :OBC).H
```

The argument N is passed as an array to create a Bose-Hubbard system with multiple boson numbers, allowing transitions between different subspaces.

3.2 `OpenQSL/src/hamiltonian.jl/RBoseHubbard`

The function

```
    RBoseHubbard(N,M,J,U)
```

creates a blockdiagonal matrix where each block is a unitary Bose-Hubbard matrix of particles on M-1 sites i.e `BlockDiagonal([BoseHubbard(i, M-1, J, U, :OBC).H for i in N:-1:0] )`. The number of particles can be anywhere from N to 0.

This is the Hamiltonian used to evolve the envrionment states in the two-point correlation functions. Furthermore `RBoseHubbard(N,M,J,U).basis` gives the basis `RBasis(N,M)`.

4 Time Evolution

4.1 `OpenQSL/src/bath.jl`

To compute the time evolution of states, such as:

$$e^{-iHt}|\psi(0)\rangle = |\psi(t)\rangle$$

we use the function:

```
function expv(a::Number, ham::BoseHubbard{T}, v::State; kwargs=())
```

This function performs the operation $\exp\{(a \cdot \texttt{ham.H})\}\,|v\rangle$ using Krylov subspace methods. The input state `v` is in the number basis, whereas the time-evolved output state is in the computational basis.

5 Two-Point Correlation Functions

5.1 OpenQSL/src/bath.jl/bath_exact

This function

```
    bath_exact(
    time1::T, time2::T, H::Vector{RBoseHubbard{S}}, i::Int, j::Int, state::State, rho; kwargs=()
) where {S, T <: Real}
```

computes the following correlation function $\langle\psi|\, b_2(s) b_2^\dagger(t) \rho_B\, |\psi\rangle$ which is evolved using `RBoseHubbard.([N+1,N],M,J,U)`. This is because the action of creation operator takes us to the space of $N+1$ particles on $M-1$ sites.

5.2 OpenQSL/src/bath.jl/bath2_exact

This function

```
    function bath2_exact(
    time1::T, time2::T, H::Vector{RBoseHubbard{S}}, i::Int, j::Int, state::State, rho; kwargs=()
) where {S, T <: Real}
```

computes the following correlation function $\langle\psi|\, b_2^\dagger(s) b_2(t) \rho_B\, |\psi\rangle$ which is evolved using `RBoseHubbard(N,M,J,U)`. This is because the action of annihilation operator takes us to the space of $N-1$ particles on $M-1$ sites which is already included in `RBoseHubbard`.

5.3 OpenQSL/src/correlators.jl/two_time_corr

This function

```
     two_time_corr(
    H::Vector{RBoseHubbard{S}},eigss,  time::Vector{T}, rho;  kwargs=()
   ) where{S, T <:Real}
```

computes the sums of the above correlation function over a basis of states:

- $$\Gamma_1(t,s) = \sum_\psi \langle\psi|\, b_2(s) b_2^\dagger(t) \rho_B\, |\psi\rangle$$
- $$\Gamma_2(t,s) = \sum_\psi \langle\psi|\, b_2^\dagger(s) b_2(t) \rho_B\, |\psi\rangle$$

and returns a list of two correlation functions $[\Gamma_1(t,s), \Gamma_2(t,s)]$

6 Partial Trace function

6.1 `OpenQSL/src/correlators.jl/partial_trace_system`

The function:

```
function partial_trace_system(init_dm, subsys_size, N, M)
```

computes the reduced density matrix by tracing out the first site of a Bose-Hubbard system with N particles and M sites. The function reduces the dimensionality of the density matrix by discarding the first site and retaining the remaining sites. It takes the following inputs:

- `init_dm`: The initial density matrix representing the full system.
- `subsys_size`: The size of the reduced subsystem after tracing out the first site. This for our case is the same size as the size of the unitary matrix.
- `N`: Number of bosons in the full system.
- `M`: Number of sites in the full system.

The function proceeds by iterating through the elements of the initial density matrix and performing the following steps:

1. It checks if the matrix element `init_dm[i,j]` is nonzero.
2. If the first elements of the basis vectors of states i and j match, the basis vectors corresponding to the first site are removed using:

   ```
   r_vec_i = deleteat!(NBasis(N,M).eig_vecs[i], 1)
   r_vec_j = deleteat!(NBasis(N,M).eig_vecs[j], 1)
   ```

3. The reduced basis indices are found in the `RBasis` set using:

   ```
   index1 = findfirst(x -> x == r_vec_i, RBasis(N,M).eig_vecs)
   index2 = findfirst(x -> x == r_vec_j, RBasis(N,M).eig_vecs)
   ```

4. The reduced density matrix is updated accordingly:

   ```
   r_dm[index1, index2] = r_dm[index1, index2] + init_dm[i,j]
   ```

Finally, the function returns the reduced density matrix `r_dm`.

6.2 `OpenQSL/src/correlators.jl/partial_trace_bath`

The function:

```
function partial_trace_bath(init_dm, N, M)
```

computes the reduced density matrix by tracing out the bath (i.e., the last $M-1$ sites) of a Bose-Hubbard system with N particles and M sites. The function reduces the dimensionality of the density matrix to represent the subsystem composed of the first site only. It takes the following inputs:

- `init_dm`: The initial density matrix representing the full system.
- `N`: The number of bosons in the full system.
- `M`: The number of sites in the full system.

6.2.1 Function Details

The function proceeds by iterating through all elements of the initial density matrix and performing the following steps:

1. It checks if the matrix element `init_dm[i,j]` is nonzero to avoid unnecessary operations.
2. If the occupation numbers corresponding to sites $2:M$ of state i and state j match, the system performs the trace:

   ```
   if NBasis(N,M).eig_vecs[i][2:end] == NBasis(N,M).eig_vecs[j][2:end]
   ```

3. The index for the reduced density matrix is determined by calculating the number of bosons at the first site:

   ```
   index1 = N+1 - NBasis(N,M).eig_vecs[i][1]
   index2 = N+1 - NBasis(N,M).eig_vecs[j][1]
   ```

4. The reduced density matrix is updated accordingly:

   ```
   r_dm[index1, index2] = r_dm[index1, index2] + init_dm[i,j]
   ```

5. Finally, the function returns the reduced density matrix in sparse format for memory efficiency:

   ```
   return sparse(r_dm)
   ```

7 Quantum Speed Limit functions

7.1 OpenQSL/src/qsl/QSL_OTOC

The structure:

```
struct QSL_OTOC{T <: Real}
```

is designed to compute the quantum speed limit (QSL) bounds for out-of-time-ordered correlators (OTOCs) in the Bose-Hubbard model. It contains the following fields:

- `state_bound`: Represents the bound obtained by analyzing the system's state evolution.
- `spectral_bound`: Represents the bound obtained by analyzing the system's spectral properties.

7.1.1 Functionality

The constructor function:

```
function QSL_OTOC(t::T, J::T, N::Int, sys_ham::RBoseHubbard{T},
                  bath_ham::Vector{RBoseHubbard{T}}, eigenvecs::Matrix{T},
                  rhob::Union{Matrix{ComplexF64},
                  BlockDiagonal{Float64, SparseMatrixCSC{Float64, Int64}}})
                  where T <: Real
```

takes in the following parameters:

- `t`: The evolution time.
- `J`: The hopping coefficient in the Bose-Hubbard model.
- `N`: The number of particles.
- `sys_ham`: The system Hamiltonian, represented as an instance of `RBoseHubbard`.
- `bath_ham`: A vector of Hamiltonians representing the environment.
- `eigenvecs`: The eigenvectors of the system.
- `rhob`: The density matrix of the bath, which may be represented as a dense or sparse block diagonal matrix.

The constructor initializes the struct by computing two key quantities:

```
state_bound = real(compute_state_bound(...))
spectral_bound = real(compute_spectral_bound(...))
```

7.1.2 State Bound Calculation

The function:

```
function compute_state_bound(t::T, J::T, N::Int, sys_ham::RBoseHubbard{T},
                             bath_ham::Vector{RBoseHubbard{T}},
                             eigenvecs::Matrix{T}, rhob::Union{Matrix{ComplexF64},
                             BlockDiagonal{Float64, SparseMatrixCSC{Float64, Int64}}})
                             where T <: Real
```

evaluates the state-based QSL bound using the following approach:

- It defines an `integrand` function that computes the product of bath correlation functions and norms of time-evolved annihilation and creation operators.
- The function performs numerical integration using the Gauss-Kronrod quadrature method (`quadgk`) to integrate over time.

7.1.3 Spectral Bound (Liouville Bound) Calculation

The function:

```
function compute_spectral_bound(t::T, J::T, N::Int, sys_ham::RBoseHubbard{T},
                                bath_ham::Vector{RBoseHubbard{T}},
                                eigenvecs::Matrix{T}, rhob::Union{Matrix{ComplexF64},
                                BlockDiagonal{Float64, SparseMatrixCSC{Float64, Int64}}})
                                where T <: Real
```

computes the spectral-based QSL bound using the following steps:

- It constructs the Liouvillian superoperator by evolving the annihilation and creation operators and computing the system-bath interactions.
- The Liouvillian operator `liouv_superop` accounts for contributions from multiple bath correlation functions.
- The spectral bound is obtained by integrating over the norm of the Liouvillian operator using `quadgk`.

8 Core Utility Functions

8.1 `OpenQSL/src/qsl/time_evol_state`

The function:

```
function time_evol_state(rho::SparseMatrixCSC{ComplexF64, Int64},
                         bh::BoseHubbard{T}, time::T) where T<: Number
```

performs time evolution of a density matrix `rho` under the Hamiltonian `bh.H` over a given `time`. The function returns the evolved density matrix:

$$\rho(t) = U\rho U^\dagger$$

where $U = e^{-iHt}$.

8.2 `OpenQSL/src/qsl/interaction_picture`

The function:

```
function interaction_picture(basis, U, rho)
```

transforms a density matrix `rho` to the interaction picture using the basis states and interaction energy term. The function applies the transformation:

$$\rho_{int}(i,j) = \rho(i,j) \cdot e^{i\frac{U}{2}n_1(n_1-1)} \cdot e^{-i\frac{U}{2}n_2(n_2-1)}$$

where n_1 and n_2 are the occupation numbers of the basis vectors.

8.3 `OpenQSL/src/qsl/renyi_entropy`

The function:

```
function renyi_entropy(rho)
```

computes the second-order Rényi entropy of the density matrix ρ, which is defined as:

$$S_2(\rho) = -\log\bigl(\mathrm{Tr}(\rho^2)\bigr)$$

where the trace of ρ^2 is calculated and returned as the entropy measure.

8.4 `OpenQSL/src/qsl/time_evol_jump_norm`

The function:

```
function time_evol_jump_norm(time::T, op, ham::RBoseHubbard{T}) where T<:Real
```

computes the norm of a time-evolved jump operator in the system. The operator evolution follows:

$$\mathcal{O}(t) = U^\dagger \mathcal{O} U$$

where $U = e^{-iHt}$. The function returns the infinity norm of the evolved operator.

8.5 `OpenQSL/src/qsl/time_evol_jump`

The function:

```
function time_evol_jump(time::T, op, ham::RBoseHubbard{T}) where T<:Real
```

performs the same operation as `time_evol_jump_norm`, but returns the full evolved operator rather than its norm.

8.6 `OpenQSL/src/qsl/limit_dm`

he function:

```
function limit_dm(rho::SparseMatrixCSC{T, Int64}, N::Int, M::Int)
                  where {T<:Number}
```

This function reduces the dimensionality of a sparse density matrix ρ to a smaller subspace using a predefined basis. It extracts the necessary elements from the input density matrix and maps them to a lower-dimensional representation.

Parameters:

- `rho` – The input density matrix in sparse format, of type `SparseMatrixCSC`.
- `N` – The number of particles in the system.
- `M` – The number of sites in the system.

Procedure:

1. Extract nonzero elements from the sparse density matrix using the `findnz` function.
2. Determine the dimensions of the reduced basis using the function `NBasis(N, M).dim`.
3. Initialize an empty matrix `limit_rho` to store the reduced density matrix.
4. Loop through the nonzero elements and map their indices to the new basis using:
   - `get_index(NBasis(N, M), tensor_basis(N, M).eig_vecs)`
5. If valid indices are found, update the corresponding matrix element.
6. Handle exceptions where indices are not found, allowing the loop to continue execution.

Returns:

- A reduced density matrix `limit_rho` in sparse format.

Error Handling:

If an error occurs during index retrieval, it is caught and ignored to allow the function to continue execution. Any unexpected errors are rethrown for further debugging.

8.7 `OpenQSL/src/qsl/create_annihilation_creation_descending`

The function:

```
function create_annihilation_creation_descending(N::Int)
```

constructs the annihilation (`a`) and creation (`adag`) operators for a bosonic system of size $N + 1$. The matrix elements are defined as:

$$a_{n,n+1} = \sqrt{n}$$

$$a^\dagger_{n+1,n} = \sqrt{n+1}$$

These matrices represent the standard ladder operators in quantum mechanics.

8.8 `OpenQSL/src/qsl/number_quench`

The function:

```
function number_quench(N,M)
```

constructs a diagonal number operator for the Bose-Hubbard model, where the expectation values correspond to the occupation number of the first site in the system.

It iterates through the basis states and assigns the particle number of the first site to each element in the matrix, normalizing the resulting matrix by its trace to ensure a valid density matrix.

9 Thermal Correlation Functions

9.1 `OpenQSL/thermal_correlator.jl`

The function:

```
function thermal_corr(beta::T, H::Vector{BoseHubbard{S}}, time::T, partition::T; kwargs=())
```

returns the values `[trsum1, trsum2]`, which are computed as follows:

- $$\texttt{trsum1} = \frac{1}{Z} \sum_{\psi \in \text{eigenvecs}} e^{-\beta E_\psi} \langle b_i(t - t = 0) b_i^\dagger(t) \rangle_\psi$$

- $$\texttt{trsum2} = \frac{1}{Z} \sum_{\psi \in \text{eigenvecs}} e^{-\beta E_\psi} \langle b_i^\dagger(t - t = 0) b_i(t) \rangle_\psi$$