Quantum Correlation Analysis in Many-Body Systems

Unnati Akhouri

QuNet Bootcamp session 3

June 5, 2025

Introduction: Why Study Correlation Spread?

- Quantum correlations are fundamental resources in quantum information
- Understanding correlation spread is crucial for:
  - Quantum thermalization
  - Entanglement distribution patterns
  - Quantum algorithm optimization
- Network analysis provides tools to visualize correlation structures
- Goal: Develop frameworks to analyze correlation dynamics and identify quantum resources

Agenda

1. Data Extraction
2. Correlation Analysis
3. Network Construction
4. Network Metrics
5. Multi-Network Comparison
6. Temporal Dynamics
7. Visualization

Data Extraction: Two-Qubit Density Matrices

```
def get_2_qbit_dms(data, n_qubits,
                   connectivity, update_rule):
    basis = ket.canonical_basis(2)
    result = []

    def to_tuple(string):
        tuple_elements = string.strip('()').split(',')
        return tuple(int(elem.strip())
                     for elem in tuple_elements)

    for trial in data[f'{n_qubits} qubits']:
        seed = trial.split(' ')[-1]
        dat = dict(data[...]['two_qubit_dms'])
        dat = {int(k.split('(')[0]): dat[k]
               for k in dat}
        result.append(dat)
    return np.array(result)
```

Purpose:

- Extract two-qubit density matrices from simulation data
- Organize by qubits, connectivity, rules, seeds
- Return structured array for analysis

Observable Extraction: What We Calculate

Extracted Observables:

- Single-qubit $\langle \sigma_z \rangle$ expectation values
- Two-qubit $\sigma_z \otimes \sigma_z$ correlations
- Connected correlations: $C_{ij} = \langle \sigma_z^i \sigma_z^j \rangle - \langle \sigma_z^i \rangle \langle \sigma_z^j \rangle$

Mathematical Foundation:

$$\langle \sigma_z^i \rangle = \mathrm{Tr}(\rho_{ij} \sigma_z \otimes I) \tag{1}$$

$$\langle \sigma_z^j \rangle = \mathrm{Tr}(\rho_{ij} I \otimes \sigma_z) \tag{2}$$

$$C_{ij} = \mathrm{Tr}(\rho_{ij} \sigma_z \otimes \sigma_z) - \langle \sigma_z^i \rangle \langle \sigma_z^j \rangle \tag{3}$$

Observable Extraction: Implementation

```
def extract_observables(two_qubit_dms, n_qubits):
    # Pauli Z matrix
    pauli_z = np.array([[1, 0], [0, -1]])
    pauli_z_tensor = np.kron(pauli_z, pauli_z)

    sz = np.zeros((n_trials, n_timesteps, n_qubits))
    corr = np.zeros((n_trials, n_timesteps, n_qubits, n_qubits))

    for trial in range(n_trials):
        for t in range(n_timesteps):
            for pair, dm in two_qubit_dms[trial, t].items():
                i, j = pair
                rho = dm.data

                # Calculate observables
                sz_i = np.trace(rho @ np.kron(pauli_z, I))
                corr_ij = np.trace(rho @ pauli_z_tensor)

    return sz.real, corr.real
```

Returns: *sz*: (trials, timesteps, qubits), *corr*: (trials, timesteps, qubits, qubits)

Correlation Analysis: Statistical Framework

Analysis Goals:

- Analyze correlation magnitude distributions
- Calculate statistics: mean, median, max, percentiles
- Generate automatic thresholds
- Compare multiple datasets

Threshold Recommendations:

- Conservative: 75th percentile
- Moderate: Mean value
- Stringent: 90th percentile

Correlation Analysis: Implementation

```
def plot_multiple_correlation_distributions(correlation_datasets, labels, colors):
    # Calculate statistics
    means = np.array([np.mean(vals) for vals in off_diag_values])
    percentile_75 = np.array([np.percentile(vals, 75)
                             for vals in off_diag_values])
    percentile_90 = np.array([np.percentile(vals, 90)
                             for vals in off_diag_values])

    # Create recommended thresholds
    recommended_thresholds = [
        overall_stats['p75'],  # Conservative
        overall_stats['mean'], # Moderate
        overall_stats['p90']   # Stringent
    ]

    # Plot histogram with KDE
    sns.histplot(all_corrs, bins=bins, kde=True)
```

Network Construction: Mutual Information

Process:

1. Calculate mutual information for each qubit pair
2. Filter values below threshold (10^{-6})
3. Create adjacency matrices
4. Generate time series of networks
5. Compute ensemble averages

Output:

- Shape: (trials, timesteps, qubits, qubits)
- Symmetric matrices (undirected networks)
- Time series of network snapshots

Network Construction: Code

```
def mutual_info_dicts(twoQdms, trial_index):
    mutual_info_list = []
    for time_step in twoQdms[trial_index]:
        mutual_info = measure.mutual_information_of_every_pair_dict(time_step)
        # Filter out values below precision threshold
        filtered_mutual_info = {k: v if v >= 1e-6 else 0
                                for k, v in mutual_info.items()}
        mutual_info_list.append(filtered_mutual_info)
    return mutual_info_list

def create_adjacency_matrix_two_dim(two_point_dict, num_nodes):
    adjacency_matrix = [[0] * num_nodes for _ in range(num_nodes)]
    for (node1, node2), value in two_point_dict.items():
        adjacency_matrix[node1][node2] = value
        adjacency_matrix[node2][node1] = value
    return adjacency_matrix
```

Network Metrics: Definitions

Global Clustering Coefficient:

$$C = \frac{\mathrm{Tr}(A^3)}{\sum_{i,j}(A^2)_{ij}} \tag{4}$$

Disparity Measure:

$$D = \frac{1}{N}\sum_{i}\frac{\sum_j A_{ij}^2}{(\sum_j A_{ij})^2} \tag{5}$$

Interpretation:

- Clustering: Local connectivity density
- Disparity: Weight distribution heterogeneity
- High disparity = few strong connections

Network Metrics: Implementation

```
def clustering_coeff(adjacency_matrix_list):
    C_list = []
    for adj_mat in adjacency_matrix_list:
        adj_mat = np.array(adj_mat)
        M_sq = adj_mat @ adj_mat
        sum_of_M_sq = np.sum(M_sq)
        M_cube = adj_mat @ adj_mat @ adj_mat
        M_cube_trace = np.trace(M_cube)
        if sum_of_M_sq == 0:
            C_list.append(0)
        else:
            C_list.append(M_cube_trace / sum_of_M_sq)
    return np.array(C_list)

def disparity_function(adjacency_matrix_list, N):
    D_list = []
    for adj_mat in adjacency_matrix_list:
        M_row_sum_squared = np.sum(adj_mat, axis=1)**2
        M_row_sum_of_squared_elements = np.sum(adj_mat**2, axis=1)
        Di = np.where(M_row_sum_squared != 0,
                      M_row_sum_of_squared_elements / M_row_sum_squared, 0)
        D_list.append(np.sum(Di) / N)
    return np.array(D_list)
```

Multi-Network Analysis

Comparative Framework:

- Analyze multiple temporal networks simultaneously
- Generate comparative metrics tables
- Consistent visualization scaling
- Edge weight distribution analysis

Computed Metrics:

- Time-averaged clustering and disparity
- Mean timestep clustering and disparity
- Network density
- Average path length

Representative Networks

Construction Method:

- Use late-time averaging (last 20-25% of simulation)
- Global color scaling for comparison
- Spring layout for optimal positioning

Visualization:

- Node color = Disparity values
- Edge thickness = Connection strength
- Global colorbar across networks
- Statistical metrics displayed below graphs

Temporal Dynamics: Correlation Growth

Analysis:

- Track total correlation: $\sum_{a,b} |C^{xx}_{ab}(\ell)|/4$
- Ensemble averaging across trials
- Identify spreading phases
- Mark critical time points

Features:

- Multi-dataset comparison
- Different initial conditions
- Various update rules
- Statistical error estimation

Temporal Dynamics: Implementation

```
def plot_mean_correlation_sums(corrs_all_datas):
    for i, corr_per_ic in enumerate(corrs_all_datas):
        for j, corr_per_rule_in_ic in enumerate(corr_per_ic):
            trial_data = corr_per_rule_in_ic[1]
            n_timesteps = len(trial_data[0])

            # Calculate mean sum for each time step across trials
            mean_correlation_sums = []
            for t in range(n_timesteps):
                step_sums = [np.sum(np.abs(trial[t]))
                           for trial in trial_data]
                mean_correlation_sums.append(np.mean(step_sums))

            x = np.arange(1, n_timesteps + 1)
            axes[i].plot(x, mean_correlation_sums, marker='o', label=f'R{j+1}')

            # Mark critical time point
            axes[i].axvline(x=10, color='red', linestyle='--')
```

Network Visualization

Consistent Framework:

- Uniform edge thickness for clarity
- Color-coded edge weights with global scaling
- Circular layout for systematic comparison
- Multiple graph comparison in single figure

Technical Implementation:

- NetworkX for graph construction
- matplotlib LineCollection for edge rendering
- Global min/max weight normalization
- Consistent axis limits

Conclusions and Takeaways

Framework Achievements:

- Complete toolkit for quantum correlation analysis
- From density matrices to network insights
- Statistical rigor with ensemble averaging
- Multi-scale analysis capabilities

Key Insights:

- Correlation patterns reveal thermalization mechanisms
- Network metrics distinguish dynamical phases
- Representative networks capture structural features
- Temporal dynamics show scaling behaviors