Many-body localization in a random Heisenberg chain

Kirill Teslenko

(Dated: July 22, 2024)

In this project, we investigate many-body localization (MBL) in a random Heisenberg chain using the Lanczos algorithm for exact diagonalization. The Hamiltonian of the system is constructed to include both interaction terms and disorder, which is essential for the study of MBL. By employing the Lanczos method, we efficiently approximate the time evolution of the system, which is critical for understanding the dynamics of quantum states in disordered systems. We start with the Néel state as the initial condition and compute the time evolution under the Heisenberg Hamiltonian with a random field. The study involves simulating the system for various disorder strengths and analyzing the evolution of the spin imbalance and entanglement entropy over time. Our results demonstrate the effectiveness of the Lanczos algorithm in handling large Hilbert spaces, providing insights into the localization properties of the system. Furthermore, We discuss the implementation details, including the simulation of the hamiltonian and the initial Néel state, and provide a Python codebase for reproducibility. The outcomes of this project contribute to the broader understanding of MBL phenomena and offer a robust computational approach for future research in quantum many-body systems.

I. INTRDUCTION

Understanding the dynamics of quantum many-body systems under the influence of disorder is crucial for exploring phenomena such as many-body localization (MBL). MBL, a phase where interacting particles remain localized due to disorder and prevents thermalization. In this study, we explore the temporal evolution of half-chain entanglement entropy and imbalance in a disordered Heisenberg chain, aiming to elucidate the effects of disorder on these key properties.

II. MODEL AND METHODS

A. Hamiltonian and Initial State

We consider a one-dimensional Heisenberg chain of length $L = 14$ with periodic boundary conditions. The Hamiltonian is given by

$$H = J \sum_{i=1}^{L} \mathbf{S}_i \cdot \mathbf{S}_{i+1} - \sum_{i=1}^{L} h_i S_i^z,$$

[1] where J is the interaction strength, $\mathbf{S}_i$ are spin-1/2 operators, and h_i are random fields uniformly distributed in $[-W, W]$. The initial state is chosen as the Néel state.

B. Lanczos Diagonalization and Time Evolution

The time evolution of the system is computed using the Lanczos algorithm[2] to obtain a tridiagonal representation of the Hamiltonian, followed by the application of the exponential of the resulting matrix. This approach allows efficient computation of the wavefunction at different time steps. The time evolution operator is applied iteratively, updating the wavefunction at each step, and the Krylov subspace[3] is reconstructed at each iteration.

C. Observables

We focus on two key observables: the half-chain entanglement entropy and the imbalance. The half-chain entanglement entropy is computed from the reduced density matrix of half the chain, while the imbalance is defined as the difference in spin expectation values between odd and even sites:

$$\text{Imbalance} = \langle \sum_{i\,\text{odd}} S_i^z \rangle - \langle \sum_{i\,\text{even}} S_i^z \rangle.$$

III. RESULTS

The selection of an appropriate time scale to represent the time evolution of half-chain entanglement entropy and imbalance has been the subject of considerable investigation. Our findings indicate that the entanglement entropy exhibits seemingly random fluctuations initially, dependent on the disorder strength W, before stabilizing. In contrast, the imbalance initially remains constant and subsequently decreases, displaying a "cardiogram-like" pattern. As shown in Figure 1, the shift in behavior moves to higher time values with decreasing disorder strength. We used $W = 1.0$ as a benchmark, where the change in behavior was observed after approximately 10 seconds. The simulation results for $W = 0.01$, 0.1, 1.0, 5.0 and 10 are presented below. The total simulation time is 1000 seconds with a time step of 1 second, ensuring that $\Delta t/J \approx 1$.

The increase in imbalance values corresponds well with the rapid rise in entropy; however, the pattern of the imbalance does not align with any known mathematical model. Repeated simulation with the same starting parameters has shown (see B.7), that the imbalance pattern is indeed not defined through initial values.

Interestingly, after evolving the system for long times $t > 10^4$ seconds (see B.6), a decrease in entropy values is observed. This nonphysical behavior suggests that

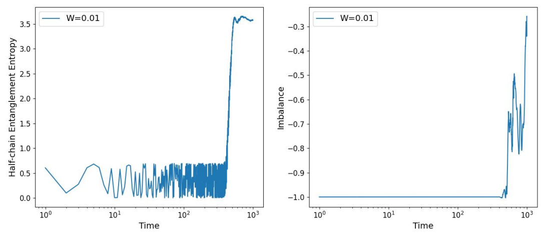

FIG. 1: Simulation results for $W = 0.01$

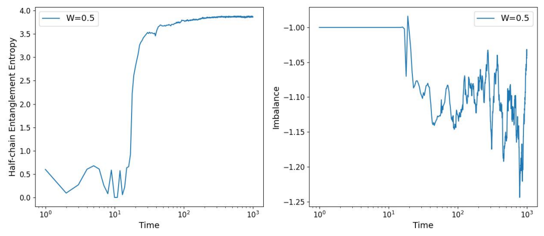

FIG. 2: Simulation results for $W = 0.5$

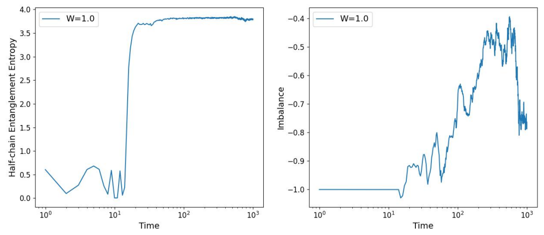

FIG. 3: Simulation results for $W = 1.0$

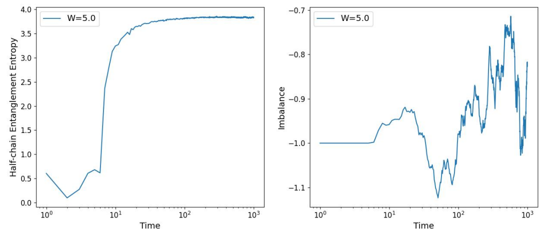

FIG. 4: Simulation results for $W = 5$

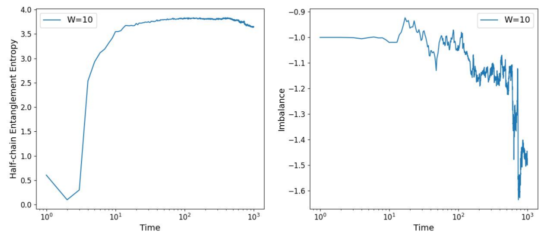

FIG. 5: Simulation results for $W = 10$

the simulation may be encountering technical limitations, which prevent it from accurately representing the system at very large times.

For large values of the disorder strength, predictably, no well-defined constant region is observed.

IV. CONCLUSION

The results define specific behavior patterns of the selected observables, thereby establishing the conditions under which the simulation yields physically relevant data and identifying its parametric limitations. Overall, our findings provide valuable insights into the effects of disorder on the entanglement entropy and imbalance in quantum many-body systems, contributing to a deeper understanding of many-body localization and the conditions under which it manifests.

V. ACKNOWLEDGMENTS

Many thanks to tutors Anton Romen, Marco Lastres, Philip Zechmann, and Raúl Morral Yepes for their support during the weekly tutorials. I also extend my gratitude to Professors Michael Knap and Frank Pollmann for their theoretical guidance in the lectures.

[1] J. H. Bardarson, F. Pollmann, and J. E. Moore, Unbounded growth of entanglement in models of many-body localization, Physical Review Letters **109**, 10.1103/physrevlett.109.017202 (2012).
[2] J. Schnack, J. Richter, and R. Steinigeweg, Accuracy of the finite-temperature lanczos method compared to simple typicality-based estimates, Phys. Rev. Res. **2**, 013186 (2020).
[3] Y. Saad, Analysis of some krylov subspace approximations to the matrix exponential operator, SIAM Journal on Numerical Analysis **29**, 209 (1992).

Appendix A: Code listing

```python
import numpy as np
import scipy
from scipy import sparse
import scipy.sparse.linalg
import matplotlib.pyplot as plt
from datetime import datetime

Id = sparse.csr_matrix(np.eye(2))
Sx = sparse.csr_matrix([[0., 1.], [1., 0.]])
Sz = sparse.csr_matrix([[1., 0.], [0., -1.]])
Splus = sparse.csr_matrix([[0., 1.], [0., 0.]])
Sminus = sparse.csr_matrix([[0., 0.], [1., 0.]])


def singlesite_to_full(op, i, L):
    op_list = [Id]*L  # = [Id, Id, Id ...] with L entries
    op_list[i] = op
    full = op_list[0]
    for op_i in op_list[1:]:
        full = sparse.kron(full, op_i, format="csr")
    return full


def gen_sx_list(L):
    return [singlesite_to_full(Sx, i, L) for i in range(L)]


def gen_sz_list(L):
    return [singlesite_to_full(Sz, i, L) for i in range(L)]


def gen_hamiltonian(sx_list, sz_list, g, J=1.):
    L = len(sx_list)
    H = sparse.csr_matrix((2**L, 2**L))
    for j in range(L):
        H = H - J *( sx_list[j] * sx_list[(j+1)%L])
        H = H - g * sz_list[j]
    return H


def lanczos(psi0, H, N=200, stabilize=False):
    """Perform a Lanczos iteration building the tridiagonal matrix T and ONB of the Krylov space.
    """
    if psi0.ndim != 1:
        raise ValueError("psi0 should be a vector, "
                         "i.e., a numpy array with a single dimension of len 2**L")
    if H.shape[1] != psi0.shape[0]:
        raise ValueError("Shape of H doesn't match len of psi0.")
    psi0 = psi0/np.linalg.norm(psi0)
    vecs = [psi0]
    T = np.zeros((N, N))
    psi = H @ psi0  # @ means matrix multiplication
    # and works both for numpy arrays and scipy.sparse.csr_matrix
    alpha = T[0, 0] = np.inner(psi0.conj(), psi).real
    psi = psi - alpha* vecs[-1]
    for i in range(1, N):
        beta = np.linalg.norm(psi)
        if beta  < 1.e-13:
            print("Lanczos terminated early after i={i:d} steps:"
                  "full Krylov space built".format(i=i))
            T = T[:i, :i]
            break
        psi /= beta
        # note: mathematically, psi should be orthogonal to all other states in `vecs`
        if stabilize:
            for vec in vecs:
                psi -= vec * np.inner(vec.conj(), psi)
            psi /= np.linalg.norm(psi)
        vecs.append(psi)
        psi = H @ psi - beta * vecs[-2]
        alpha = np.inner(vecs[-1].conj(), psi).real
        psi = psi - alpha * vecs[-1]
        T[i, i] = alpha
        T[i-1, i] = T[i, i-1] = beta
    return T, vecs


def colorplot(xs, ys, data, **kwargs):
    """Create a colorplot with matplotlib.pyplot.imshow.

    Parameters
    ----------
    xs : 1D array, shape (n,)
        x-values of the points for which we have data; evenly spaced
    ys : 1D array, shape (m,)
        y-values of the points for which we have data; evenly spaced
    data : 2D array, shape (m, n)
        ``data[i, j]`` corresponds to the points ``(xs[i], ys[j])``
    **kwargs :
        additional keyword arguments, given to `imshow`.
    """
    data = np.asarray(data)
    if data.shape != (len(xs), len(ys)):
        raise ValueError("Shape of data doesn't match len of xs and ys!")
    dx = (xs[-1] - xs[0])/(len(xs)-1)
    assert abs(dx - (xs[1]-xs[0])) < 1.e-10
    dy = (ys[-1] - ys[0])/(len(ys)-1)
    assert abs(dy - (ys[1]-ys[0])) < 1.e-10
    extent = (xs[0] - 0.5 * dx, xs[-1] + 0.5 * dx,  # left, right
              ys[0] - 0.5 * dy, ys[-1] + 0.5 * dy)  # bottom, top
    kwargs.setdefault('aspect', 'auto')
    kwargs.setdefault('interpolation', 'nearest')
    kwargs.setdefault('extent', extent)
    # convention of imshow: matrix like data[row, col] with (0, 0) top left.
    # but we want data[col, row] with (0, 0) bottom left -> transpose and invert y axis
    plt.imshow(data.T[::-1, :], **kwargs)


# Additional functions to generate Hamiltonian for Neel state


def plot_E_vs_LanzcosIter(T):
    # Plot the results
    E = np.linalg.eigvalsh(T)
    Ns = np.arange(10, len(T))
    plt.figure(figsize=(13, 10))
    Es = []
    for Num in Ns:
        E = np.linalg.eigvalsh(T[:Num, :Num])
        Es.append(E[:10])

    plt.plot(Ns, Es)
    #plt.ylim(np.min(Es)-0.1, np.min(Es) + 5.)
    plt.title("stabilize")
    plt.xlabel("Lanczos iteration $N$")
    plt.ylabel("Energies")
    current_time = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    filename = f"plot_EvsIterat_{current_time}.jpg"
    plt.savefig(filename)
    plt.show()

    return None
def generate_neel_state(L):

    length = 2 ** L
    vector = np.array([(i % 2) for i in range(length)])
    neel_state = np.roll(vector, 1)

    return neel_state / np.linalg.norm(neel_state)

def generate_random_field(L, W):
    return np.random.uniform(-W, W, size=L)

def gen_heisenberg_hamiltonian(sz_list, J=1., W=1.):
    L = len(sz_list)
    H = sparse.csr_matrix((2**L, 2**L))
    h = generate_random_field(L, W)
    for i in range(L):
        H += J * sz_list[i] @ sz_list[(i+1) % L]
        H -= h[i] * sz_list[i]
    return H

def time_evolve(psi0, H, t, N=200):
    """Evolve the state psi0 under Hamiltonian H for time t using Lanczos algorithm."""
    T, vecs = lanczos(psi0, H, N)
    eigvals, eigvecs = np.linalg.eigh(T)
    exp_T = np.diag(np.exp(-1j * eigvals * t))
    evolved_vecs = eigvecs @ exp_T @ eigvecs.T
    psi_t = np.zeros(psi0.shape, dtype=np.complex128)
    for i, vec in enumerate(vecs):
        psi_t += evolved_vecs[0, i] * vec
    return psi_t

def plot_time_evolve(T):
    eigvals, eigvecs = np.linalg.eigh(T)
    plt.plot(eigvals)
    plt.xlabel("Index")
    plt.ylabel("Eigenvalue")
    plt.show()
    return True

def reduced_density_matrix(psi, L):
    """Calculate the reduced density matrix for the first half of the chain."""
    psi = psi.reshape([2] * L)
    dim = 2**(L//2)
    psi = psi.transpose([i for i in range(0, L//2)] + [i for i in range(L//2, L)])
    psi = psi.reshape((dim, dim))
    rho = np.dot(psi, psi.conj().T)
    return rho

def entanglement_entropy(rho):
    """Calculate the entropy of a density matrix."""
    eigenvalues = np.linalg.eigvalsh(rho)
    entropy = -np.sum(eigenvalues * np.log(eigenvalues + 1e-12))
    return entropy


def calculate_imbalance(psi, sz_list, L):
    sz_odd = sum([sz_list[i] for i in range(L) if i % 2 == 0])
    sz_even = sum([sz_list[i] for i in range(L) if i % 2 != 0])
    sz_odd_exp = np.vdot(psi, sz_odd @ psi).real
    sz_even_exp = np.vdot(psi, sz_even @ psi).real
    imbalance = sz_odd_exp - sz_even_exp
    return imbalance

if __name__ == "__main__":
    L = 14 # System size
    W = 1. # Disorder strength
    sx_list = gen_sx_list(L)
    sz_list = gen_sz_list(L)

    H = gen_heisenberg_hamiltonian(sz_list, J=1., W=W)
    psi0 = generate_neel_state(L)
    #print(psi0)


    # Uncomment out to get plot analogous to exercise 5
    # Perform Lanczos iteration
    T, vecs = lanczos(psi0, H, N=200, stabilize=True)
    plot_E_vs_LanzcosIter(T)


    # Generate time steps
    time = list(range(1, 1001))

    # Time evolution

    fig, axs = plt.subplots(1, 2, figsize=(15, 6))

    psi_t = psi0
    plotE = []
    plotImba = []
    for t in time:
        psi_t_next = time_evolve(psi_t, H, t, N=200)

        rho = reduced_density_matrix(psi_t_next, L)
        entropy = entanglement_entropy(rho)
        plotE.append(entropy)
        print(f"Half-chain entanglement entropy at time t={t}: {entropy}")
        psi_t = psi_t_next
        imba = calculate_imbalance(psi_t, sz_list, L)
        plotImba.append(imba)
        print(f"Imbalance at time t={t}: {imba}")

    axs[0].plot(time, plotE, label=f"W={W}")
    axs[1].plot(time, plotImba, label=f"W={W}")

    axs[0].set_xscale('log')
    axs[1].set_xscale('log')

    axs[0].set_xlabel('Time', fontsize=14)
    axs[0].set_ylabel('Half-chain Entanglement Entropy', fontsize=14)
    axs[1].set_xlabel('Time', fontsize=14)
    axs[1].set_ylabel('Imbalance', fontsize=14)

    axs[0].legend(fontsize=14)
    axs[1].legend(fontsize=14)

    axs[0].tick_params(axis='both', which='major', labelsize=12)
    axs[1].tick_params(axis='both', which='major', labelsize=12)

    current_time = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    filename = f"Bplot_Entanglement_Imbalance_w001_{current_time}.jpg"
    plt.savefig(filename)
    plt.show()
```

Appendix B: Additional figures

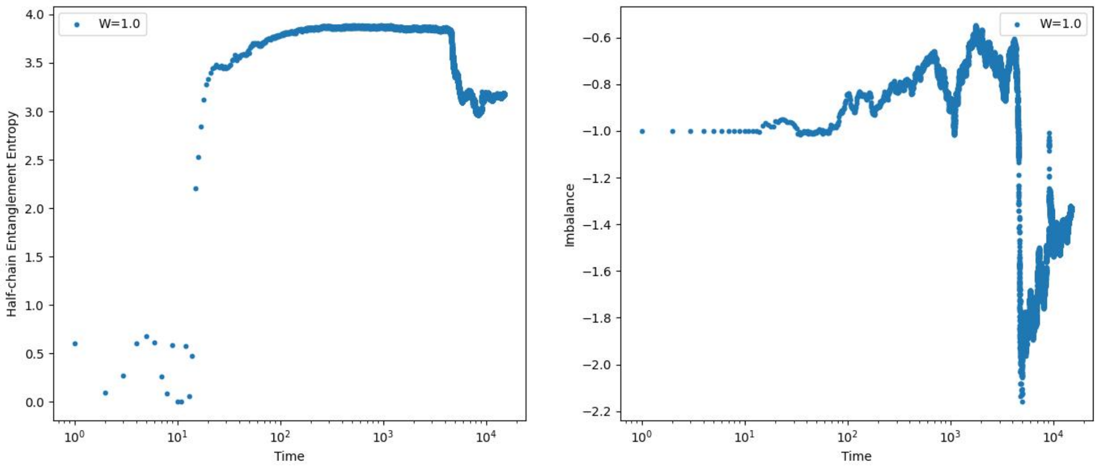

FIG. B.6: Simulation results for $W = 1.0$ over 1.5×10^4s

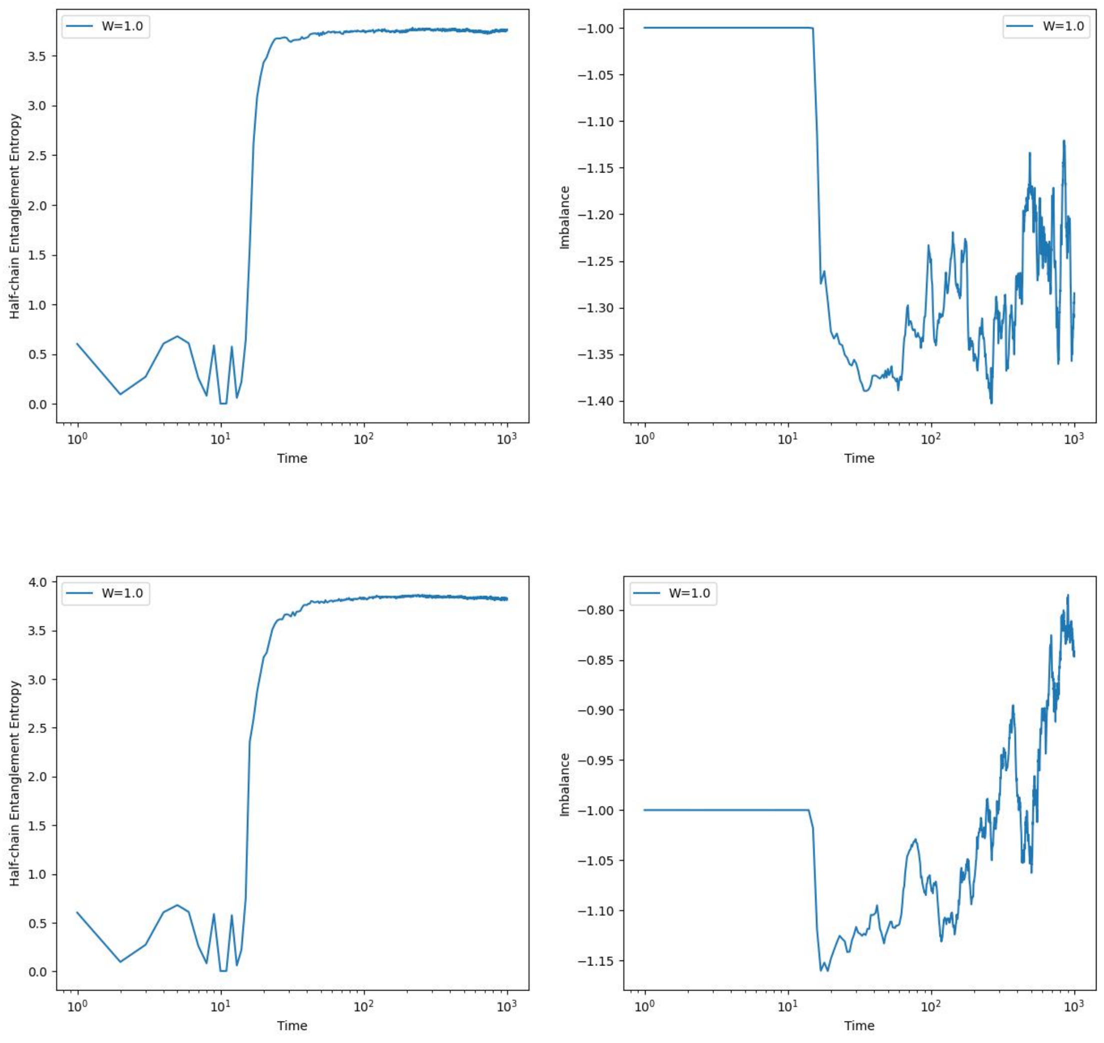

FIG. B.7: Combined simulation results with the same starting parameters