Time Evolution of Quantum Systems 2025: Exercise 2

M. Gisti, T. Luu, M. Maležič, J. Ostmeyer Hand in: 07.05.2025

Quicky: Matrix product trace

Q.1 The trace of a matrix product $tr(AB)$ is to be calculated. A and B are $n \times n$ matrices. Sketch this operation in Tensor Network notation and calculate the runtimes for the different link contraction orders. How should this operation be implemented? (1 P.)

Exact Diagonalization using Fourier Transform

H.1 Consider a single particle in a 3D cubic lattice of size $L \times L \times L$ with periodic boundary conditions. The system is governed by the tight-binding Hamiltonian

$$H = -t \sum_{\langle \mathbf{r}, \mathbf{r}' \rangle} (c_{\mathbf{r}}^\dagger c_{\mathbf{r}'} + c_{\mathbf{r}'}^\dagger c_{\mathbf{r}})$$

where $c_{\mathbf{r}}^\dagger$ and $c_{\mathbf{r}}$ are creation and annihilation operators at lattice site $\mathbf{r} = (x, y, z)$, t is the hopping amplitude, and the sum $\sum_{\langle \mathbf{r}, \mathbf{r}' \rangle}$ runs over nearest-neighbor sites in all three spatial directions.

(a) Perform a Fourier transform of the creation and annihilation operators,

$$c_{\mathbf{k}} = \frac{1}{\sqrt{L^3}} \sum_{\mathbf{r}} e^{-i\mathbf{k}\cdot\mathbf{r}} c_{\mathbf{r}} \qquad c_{\mathbf{k}}^\dagger = \frac{1}{\sqrt{L^3}} \sum_{\mathbf{r}} e^{-i\mathbf{k}\cdot\mathbf{r}} c_{\mathbf{r}}^\dagger$$

and write the Hamiltonian in momentum space. (1 P.)

(b) In the momentum space, the Hamiltonian results to be already diagonal. Determine the eigenvalues as a function of the momenta (or the energy dispersion relation). (1 P.)

Numerical Exact Diagonalization

H.2 In this exercise, you will write a small exact diagonalization code to solve a fundamental problem of quantum mechanics: the one-dimensional Heisenberg XXZ model. Consider a system of L spin-$\frac{1}{2}$ particles, the Heisenberg XXZ model has nearest-neighbor interactions,

$$H = J \sum_{j=0}^{L-1} \left(S_j^x S_{j+1}^x + S_j^y S_{j+1}^y + \delta S_j^z S_{j+1}^z \right).$$

The exact diagonalization of this model becomes computationally challenging for large system sizes due to the exponential growth of the Hilbert space.

The goal of this exercise is to diagonalize the Hamiltonian. There are many neat tools for various languages. In Python, you can represent the Hamiltonian as a sparse matrix, `scipy.sparse.csr_matrix` and diagonalize it using (a variant of) the Lanczos algorithm provided in `scipy`.

Instead of full diagonalization, the *Lanczos algorithm* provides an efficient way to find a few extremal eigenvalues for a Hermitian matrix A. The Lanczos algorithm constructs an orthonormal basis for the *Krylov subspace* generated by the matrix A and a randomly chosen initial vector v_0. The Krylov subspace is defined as:

$$\mathcal{K}_k(A, v_0) = \text{span}\{v_0, Av_0, A^2v_0, \dots, A^{k-1}v_0\}.$$

Instead of directly working with the basis $\{b, Ab, \dots\}$, the Lanczos algorithm produces an orthonormal basis $\{q_1, q_2, \dots, q_k\}$ for this subspace.

- Initialization of the algorithm:
  - Choose a starting vector q_1 with unit norm ($\|q_1\|_2 = 1$).
  - Set $\beta_0 = 0$ and $q_0 = 0$.
- Iteration ($j = 1, 2, \dots, k$):
  - Compute $w_j = Aq_j$.
  - Orthogonalize w_j with respect to the previous two Lanczos vectors:
    - ◇ $\alpha_j = q_j^T w_j$
    - ◇ $w_j = w_j - \alpha_j q_j - \beta_{j-1} q_{j-1}$
  - Compute the norm of the resulting vector:
    - ◇ $\beta_j = \|w_j\|_2$
  - If $\beta_j = 0$, the Krylov subspace is invariant, and the algorithm terminates.
  - Compute the next Lanczos vector:
    - ◇ $q_{j+1} = \frac{w_j}{\beta_j}$

After k iterations, the algorithm produces an orthonormal basis $Q_k = q_1, q_2, \dots, q_k$ and with the projection of A onto this subspace we obtain a $k \times k$ symmetric tridiagonal matrix $T_k = Q_k^T A Q_k$, in the form:

$$T_k = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{k-1} \\ & & & \beta_{k-1} & \alpha_k \end{pmatrix}.$$

The eigenvalues of this smaller tridiagonal matrix, known as *Ritz values*, are good approximations of the extremal eigenvalues of the original matrix A. This is because the Krylov subspace is built to capture the directions most *stretched* or *shrunk* by the transformation A, which correspond to the eigenvectors associated with the *largest* and *smallest* eigenvalues. As the Krylov subspace grows, the extremal eigenvalues of T_k quickly become accurate estimates of those of A. The eigenvectors of T_k (which are simpler to compute) can then approximate the eigenvectors of A.

This method is efficient because it only needs matrix-vector multiplications, saving storage and avoiding full diagonalization. In the following tasks, we will appreciate this efficiency using sparse matrices, where the main cost due to matrix-vector multiplication is much faster than that of dense matrices.

(a) Represent the spin operators by constructing the 2×2 matrices as `scipy.sparse.csr_matrix`

$$\mathbb{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad 2S^x = \sigma^x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad 2S^z = \sigma^z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \qquad \text{(1 P.)}$$

(b) Recall that σ_j^z stands for $\sigma_j^z \equiv \mathbb{1} \otimes \cdots \otimes \mathbb{1} \otimes \sigma^z \otimes \mathbb{1} \otimes \cdots \otimes \mathbb{1}$ in the full operator space, where σ^z is at the j-th position. The operator σ_j^z can be represented as a $2^L \times 2^L$ matrix. In order to represent the tensor product, use successive calls to the function `scipy.sparse.kron()`. For a given L, write a function that returns a list containing a representation of σ_j^z (as a `csr_matrix`) as the j-th entry of the list. (2 P.)

(c) Write a similar function returning the σ_j^x operators. (2 P.)

(d) Construct the Hamiltonian as a `csr_matrix` of shape $2^L \times 2^L$, where the arguments `sx_list`, `sz_list` should be the lists generated in the previous parts. (2 P.)

(e) After constructing H as a `csr_matrix`, use the function `scipy.sparse.linalg.eigsh` to obtain the ground state. The code has a similar structure:

```python
from scipy.sparse.linalg import eigsh

L = 6  # System size
H = construct_hamiltonian(L)
num_eigenvalues = 3  # Compute lowest 3 eigenvalues

eigenvalues, eigenvectors = eigsh(H, k=num_eigenvalues)
print("Lowest energy levels:", eigenvalues)
```

Compare the run time for different system sizes: $L = 6, 8, 10$. (2 P.)

(f) Compare the run time of `scipy.sparse.linalg.eigsh` with full diagonalization performed by `np.linalg.eigh`, on a dense NumPy array for $L = 12$.

(1 P.)

(g) For a fixed value of δ, compute the energy gap $\Delta E = E_1 - E_0$. (1 P.)

(h) At $T = 0$, for $\delta > 1$, the ground state is a Néel antiferromagnet phase, for $-1 < \delta < 1$ the system is in a gapless phase where quantum fluctuations destroy the long-range order. In the thermodynamical limit (for $L \to \infty$), the model at $\delta = 1$ the system undergoes a quantum phase transition between the two phases, from a gapless phase to a gapped phase. Plot the energy gap ΔE as a function of δ and observe the critical behavior. (2 P.)

Trotter-Suzuki Decomposition

H.3 The unitary time evolution operator for a closed quantum system over time t is given by

$$U(t) = e^{-iHt}.$$

(a) Define the following terms by dividing the sum over odd- and even-bonds of the Hamiltonian of the previous exercise:

- $H_{odd} = \sum_{i=1,3,5,\ldots}^{L-1} J \left(S_j^x S_{j+1}^x + S_j^y S_{j+1}^y + \delta S_j^z S_{j+1}^z \right)$
- $H_{even} = \sum_{i=2,4,6,\ldots}^{L-1} J \left(S_j^x S_{j+1}^x + S_j^y S_{j+1}^y + \delta S_j^z S_{j+1}^z \right)$.

Express the original Hamiltonian H in terms of H_{odd} and H_{even}. (1 P.)

(b) Show that, in general, the two operators do not commute, $[H_{odd}, H_{even}] \neq 0$. (1 P.)

(c) Since H_{odd} and H_{even} do not commute, $e^{-iHt} \neq e^{-iH_{odd}t} \cdot e^{-iH_{even}t}$. However, we can use the Trotter-Suzuki decomposition to approximate $U(t)$. Defining $N \in \mathbb{N}$ as the number of time steps and $\tau = t/N$, the **first-order Trotter-Suzuki approximation** is

$$U(t) = (U(\tau))^N \approx \left(e^{-iH_{odd}\tau} e^{-iH_{even}\tau}\right)^N .$$

Using the BCH expansion for $U(\tau)$, prove the first-order Trotter-Suzuki decomposition. Determine the exponents α, β in the leading order error $\mathcal{O}\left(t^\alpha \tau^\beta\right)$. (2 P.)

(d) The simplest **second-order Trotter-Suzuki decomposition** is given by:

$$U(t) = \left(e^{-iH_{odd}\frac{\tau}{2}} e^{-iH_{even}\tau} e^{-iH_{odd}\frac{\tau}{2}} + O(\tau^3)\right)^N$$

Write a Python function (or in your preferred language) that calculates the second-order Trotter-Suzuki approximation for a given Hamiltonian, time t, and fixed N. (2 P.)

(e) Construct a **fourth-order Trotter-Suzuki decomposition** using Suzuki's recursive formula and implement a Python function (or in your preferred language) to calculate this approximation. (2 P.)

(f) For a small system of $L = 6$ spins, set the interaction strengths $J = \delta = 1$. Calculate the exact time evolution operator $U(t)$ using a matrix exponentiation function from a library. (1 P.)

(g) For a fixed time t (e.g. $t = 1$), compute the second-order and fourth-order Trotter-Suzuki approximations for different numbers of time steps: $N = 1, 2, 4, 8, 16$. (1 P.)

(h) Calculate the error ϵ between the exact time evolution operator and the approximations using the operator norm (or Frobenius norm):

$$\epsilon(N) = ||U(t) - U_{approx}(t)||_F \qquad ||A||_F = \sqrt{Tr(A^\dagger A)}$$

(2 P.)

(i) Plot the error as a function of N for both the second-order and fourth-order approximations on the same plot (use a log-log scale). Compare the convergence rate of the error for the second-order and fourth-order methods and observe how the error changes as N increases. (1 P.)

(j) Discuss how the choice of the order of the Trotter-Suzuki decomposition and the number of steps N affect the accuracy of the simulation. What are the trade-offs? (1 P.)

For each relevant task, submit Python implementations and plots for different system sizes.