Simulation of Quantum Systems with Time-Evolving Block Decimation

https://github.com/jchryssanthacopoulos/quantum_information/tree/main/final_project

Quantum Information and Computing
AA 2022–23

James Chryssanthacopoulos
with collaboration with David Lange
8 April 2023

Università degli Studi di Padova

Outline

Simulating Quantum Systems

Università degli Studi di Padova

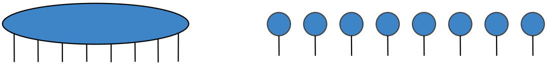
- To study a quantum system, one has to solve Schrodinger equation

$$\hat{H}\left|\Psi(t)\right\rangle = i\hbar\frac{\partial}{\partial t}\left|\Psi(t)\right\rangle$$

- One method involves direct numerical integration, where initial state is updated using time evolution operator

$$\left|\Psi(t+\Delta t)\right\rangle = e^{-i\hat{H}\Delta t}\left|\Psi(t)\right\rangle$$

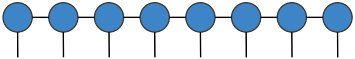
- This requires solving system of equations at every time that scales with system size, but in many-body problems, system size is **exponential** in number of sites, N
- In tensor network notation, general N-body system is shown on left. Mean-field ansatz on right greatly simplifies computation, but it ignores entanglement



How does one **preserve entanglement** while remaining **computationally tractable**?

Matrix Product States

- Matrix product states generalize mean-field ansatz to allow for entanglement between sites. Graphically,



  where bond between sites has fixed **bond dimension** χ. When $\chi = 1$, mean-field approximation recovered

- Wavefunction is given by

$$|\Psi\rangle = A_1^{\mu_1} A_{\mu_1,2}^{\mu_2} \cdots A_{\mu_{N-2},N-1}^{\mu_{N-1}} A_{\mu_{N-1},N} \, |1\,2 \cdots N\rangle$$

  where $A_{\mu_{i-1},i}^{\mu_i}$ tensors have physical dimension $i \in \{1, \ldots, d\}$ and **auxiliary dimension** $\mu_i \in \{1, \ldots, \chi\}$

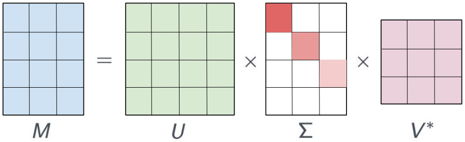
- Number of states scales like $Nd\chi^2$, which is **polynomial** in N

How does one evolve MPS in time without **breaking structure**?

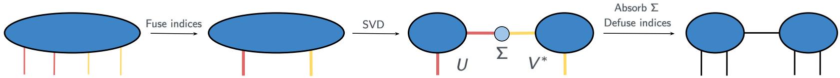
Factoring Quantum State into MPS

Università degli Studi di Padova

- An arbitrary quantum state can be factored as an MPS using matrix factorization technique called **singular value decomposition**
- SVD generalizes eigendecomposition, finding two orthonormal bases u_i, w_i and singular values σ_i such that matrix is factorized into $M = U\Sigma V^*$, with U, V unitary



- SVD can be used to successively factor an MPS using following process:



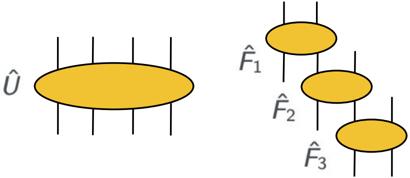
- SVD is driver behind simulating MPS quantum systems, allowing MPS structure to be preserved at each iteration. To keep bond dimension constant, singular values must be **truncated**

Time-Evolving Block Decimation

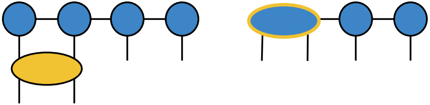
Università degli Studi di Padova

- Method for evolving quantum system while efficiently truncating large Hilbert space
- Also called **t-DMRG**, it evolves state using local gate operators and uses SVD to factorize back into MPS structure. The steps are:

① Factor time evolution operator $\hat{U} : d^N \to d^N$ into two-site gates $\hat{F}_i : d^2 \to d^2$



② Apply first gate to first two sites, contracting indices to produce new state



③ Use SVD to factor state back into MPS form, truncating to bond dimension χ

④ Repeat for each pair of neighbors and time step

Suzuki-Trotter Decomposition

- Approximate decomposition of Hamiltonian based on Baker-Campbell-Hausdorff formula that reduces time and storage complexity of applying time evolution operator
- Hamiltonian $\hat{H}$ can be decomposed into odd and even operators:

$$\hat{H} = \sum_{i} \hat{h}_{i,i+1} = \sum_{i \text{ odd}} \hat{h}_{i,i+1} + \sum_{i \text{ even}} \hat{h}_{i,i+1} \equiv \hat{H}_{\text{odd}} + \hat{H}_{\text{even}}$$

- In **first-order Suzuki-Trotter**, commutator is ignored, leading to error $\mathcal{O}(\Delta t^2)$:

$$\hat{U} = e^{-i\hat{H}\Delta t} = e^{-i\hat{H}_{\text{even}}\Delta t} e^{-i\hat{H}_{\text{odd}}\Delta t} e^{-i\left[\hat{H}_{\text{even}}, \hat{H}_{\text{odd}}\right]\Delta t^2} \approx e^{-i\hat{H}_{\text{even}}\Delta t} e^{-i\hat{H}_{\text{odd}}\Delta t} + \mathcal{O}(\Delta t^2)$$

- Higher orders can be built by continuing to factor Hamiltonian into finer time steps. For example, **second-order Suzuki-Trotter** comes with $\mathcal{O}(\Delta t^3)$ error:

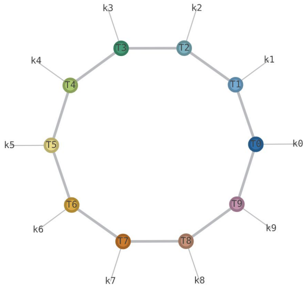
$$\hat{U} \approx e^{-i\hat{H}_{\text{even}}\Delta t/2} e^{-i\hat{H}_{\text{odd}}\Delta t} e^{-i\hat{H}_{\text{even}}\Delta t/2} + \mathcal{O}(\Delta t^3)$$

- When applied to time steps $T/\Delta t$, errors of ST1 and ST2 are $\mathcal{O}(\Delta t)$ and $\mathcal{O}(\Delta t^2)$, respectively. ST2 has lower error but requires more operations

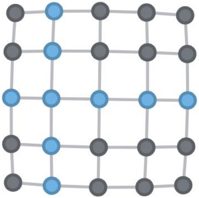
Università degli Studi di Padova

- Contains tools for working with tensors and tensor networks, including contracting, optimizing, and drawing them

`Tensor`



`TensorNetwork`



`TensorNetwork2D`

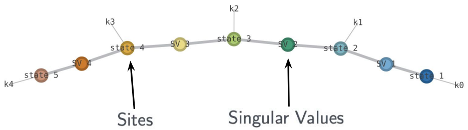
- Although it supports more complicated geometries and algorithms, only the basic `Tensor` and `TensorNetwork` classes were used

Implementing Matrix Product States

Università degli Studi di Padova

Class used in TEBD algorithm to model basic MPS structure, initializing, contracting, and computing observables from it

```python
class MatrixProductState:
    """Class representing a matrix product state with given number of states."""

    def __init__(
            self, d: int, N: int, bond_dim: int, states: Optional[List[np.array]] = None, rng_seed: Optional[int] = 0
    ):
        """Initialize the MPS.

        Args:
            d: Dimension of each state
            N: Number of states
            bond_dim: Bond dimension between states
            states: Optional states to initialize with
            rng_seed: Number to seed random number generator

        """
        self.d = d
        self.N = N
        self.bond_dim = bond_dim

        self.data = []

        if not states:
            np.random.seed(rng_seed)

            states = []
            states.append(np.random.rand(d, bond_dim))

            for i in range(1, N - 1):
                states.append(np.random.rand(bond_dim, d, bond_dim))

            states.append(np.random.rand(bond_dim, d))

        # create left-most state
        self.data.append(qtn.Tensor(states[0], inds=("k0", "i0"), tags=["state 1"]))

        for i in range(1, N - 1):
            self.data.append(
                qtn.Tensor(np.eye(bond_dim, bond_dim), inds=(f"i{2 * (i - 1)}", f"i{2 * i - 1}"), tags=[f"SV {i}"])
            )
            self.data.append(
                qtn.Tensor(states[i], inds=(f"i{2 * i - 1}", f"k{i}", f"i{2 * i}"), tags=[f"state {i + 1}"])
            )

        # create right-most state
        self.data.append(
            qtn.Tensor(np.eye(bond_dim, bond_dim), inds=(f"i{2 * (N - 2)}", f"i{2 * N - 3}"), tags=[f"SV {N - 1}"])
        )
        self.data.append(qtn.Tensor(states[N - 1], inds=(f"i{2 * N - 3}", f"k{N - 1}"), tags=[f"state {N}"]))

        self.normalize()
```



Contains methods for

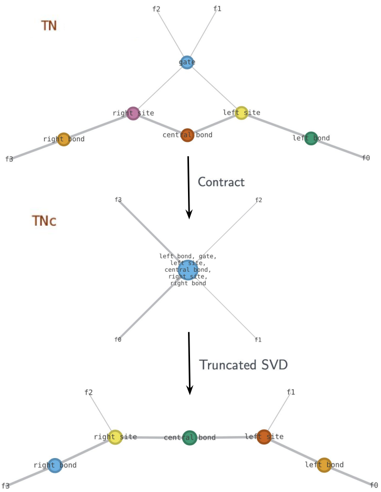
- Generating density matrix tensor
- Computing magnetization
- Calculating entropy
- Extracting wavefunction

Implementation of TEBD Algorithm

TEBD class accepts `MatrixProductState` object and implements `step` method that applies gate on every pair of states

Apply Gate

```
left_bond_T = qtn.Tensor(left_bond_data, inds=('f0', 'k1'), tags=['left bond'])
left_site_T = qtn.Tensor(left_site.data, inds=('k1', 'k2', 'k3'), tags=['left site'])
central_bond_T = qtn.Tensor(central_bond.data, inds=('k3', 'k4'), tags=['central bond'])
right_site_T = qtn.Tensor(right_site.data, inds=('k4', 'k5', 'k6'), tags=['right site'])
right_bond_T = qtn.Tensor(right_bond_data, inds=('k6', 'f3'), tags=['right bond'])
gate_T = qtn.Tensor(gate, inds=('f1', 'f2', 'k2', 'k5'), tags=['gate'])

# contract with gate
TN = left_bond_T & gate_T & left_site_T & central_bond_T & right_site_T & right_bond_T
TNc = TN ^ ...

# perform SVD
nshape = [self.d * left_site.data.shape[0], self.d * right_site.data.shape[2]]
utemp, stemp, vhtemp = LA.svd(TNc.data.reshape(nshape), full_matrices=False)

# truncate to reduced dimension
chitemp = min(self.bond_dim, len(stemp))
utemp = utemp[:, range(chitemp)].reshape(left_site.data.shape[0], self.d * chitemp)
vhtemp = vhtemp[range(chitemp), :].reshape(chitemp * self.d, right_site.data.shape[2])

# remove environment weights to form new MPS tensors A and B
left_site.modify(data=(LA.inv(left_bond_data) @ utemp).reshape(left_site.data.shape[0], self.d, chitemp))
right_site.modify(data=(vhtemp @ LA.inv(right_bond_data)).reshape(chitemp, self.d, right_site.data.shape[2]))
central_bond.modify(data=np.diag(stemp[range(chitemp)] / LA.norm(stemp[range(chitemp)])))
```



Ground State Energy

Università degli Studi di Padova

References

S. Paeckel, T. Köhler, A. Swoboda, S. R. Manmana, U. Schollwöck, and C. Hubig, "Time-evolution Methods for Matrix-product States," Annals of Physics, vol. 411, December 2019, 167998

J. Gray, "QUIMB: A Python Library for Quantum Information and Many-body Calculations," Journal of Open Source Software, vol. 3, no. 29, 2018