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
in collaboration with David Lange
8 April 2023

Università degli Studi di Padova

Outline

Theory

Simulating Quantum Systems

UNIVERSITÀ DEGLI STUDI DI PADOVA

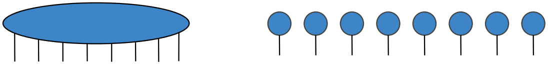
- To study a quantum system, one has to solve Schrodinger equation

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

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

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

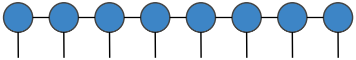
- This requires solving system of equations at every time that scales with system size, but in many-body problems, system size is **exponential** in number of sites, N
- In tensor network notation, general N-body system is shown on left. Mean-field ansatz on right greatly simplifies computation, but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

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

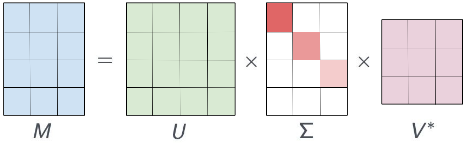
where $A_{\mu_{i-1},i}^{\mu_i}$ tensors have physical dimension $i \in \{1, \dots, d\}$ and **auxiliary dimension** $\mu_i \in \{1, \dots, \chi\}$

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

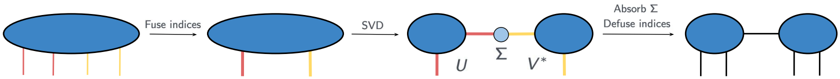
How does one evolve MPS in time without **breaking structure**?

Factoring Quantum State into MPS

- 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:

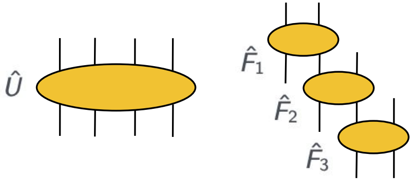


- SVD is driver behind simulating MPS quantum systems, allowing MPS structure to be preserved at each iteration. To keep bond dimension constant, singular values must be **truncated**

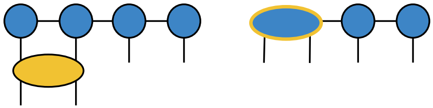
Time-Evolving Block Decimation

- 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:

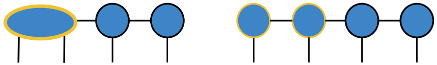
① Factor time evolution operator $\hat{U} : d^N \to d^N$ into two-site gates $\hat{F}_i : d^2 \to d^2$



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:

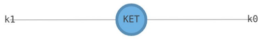
$$\hat{U} \approx e^{-i\hat{H}_{\text{even}}\Delta t/2} e^{-i\hat{H}_{\text{odd}}\Delta t} e^{-i\hat{H}_{\text{even}}\Delta t/2} + \mathcal{O}(\Delta t^3)$$

- 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

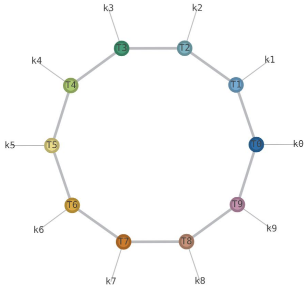
Code Development

QUIMB: Quantum Computing in Python

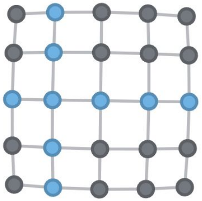
- Contains tools for working with tensors and tensor networks, including contracting, optimizing, and drawing them



`Tensor`



`TensorNetwork`



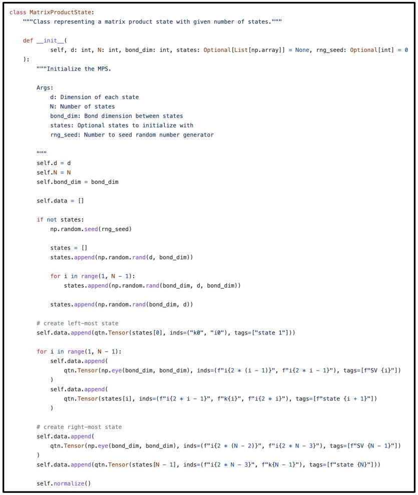
`TensorNetwork2D`

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

Implementing Matrix Product States

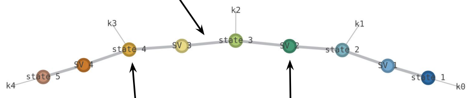
Università degli Studi di Padova

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()
```

Edge width indicates bond dimension



Sites

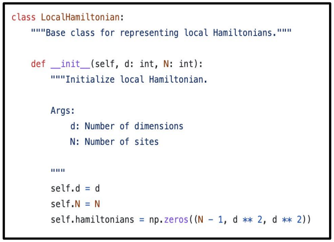
Singular Values

Contains methods for

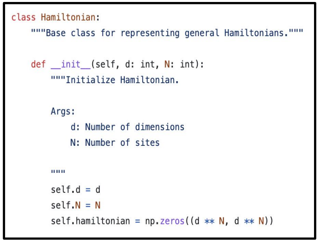
- Generating density matrix tensor
- Computing magnetization
- Calculating entropy
- Extracting wavefunction

Implementing Hamiltonians

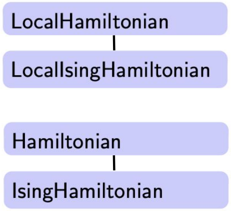
Local and global Hamiltonians implemented to evolve state and compute energy



```python
class LocalHamiltonian:
    """Base class for representing local Hamiltonians."""

    def __init__(self, d: int, N: int):
        """Initialize local Hamiltonian.

        Args:
            d: Number of dimensions
            N: Number of sites

        """
        self.d = d
        self.N = N
        self.hamiltonians = np.zeros((N - 1, d ** 2, d ** 2))
```

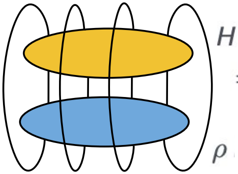


```python
class Hamiltonian:
    """Base class for representing general Hamiltonians."""

    def __init__(self, d: int, N: int):
        """Initialize Hamiltonian.

        Args:
            d: Number of dimensions
            N: Number of sites

        """
        self.d = d
        self.N = N
        self.hamiltonian = np.zeros((d ** N, d ** N))
```

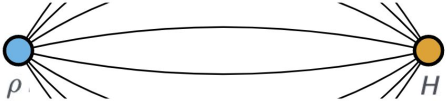


Global Hamiltonian is contracted with density matrix to compute energy

$E = \mathrm{Tr}(\rho H) =$



$=$



Implementing TEBD Algorithm

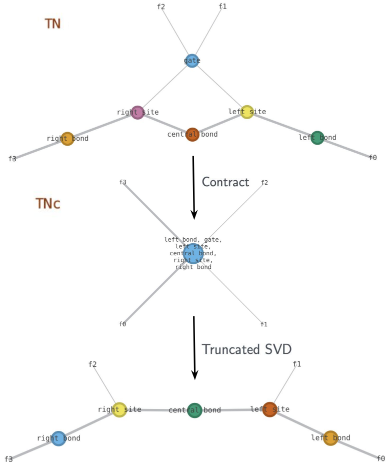
UNIVERSITÀ DEGLI STUDI DI PADOVA

TEBD class accepts `MatrixProductState` and `Hamiltonian` objects and implements `step` method that applies gate on every pair of states from left to right

Apply Gate

Inputs:
`gate`, `left_site`, `right_site`, `left_bond`, `central_bond`, `right_bond`

```
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
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)])))
```



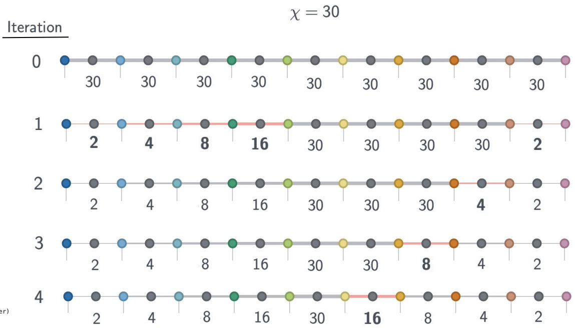
Running TEBD

- Main execution function accepts model, model parameters, and other run parameters
- Bond dimension across chain evolves over first few iterations due to SVD
- Time complexity scales like $\mathcal{O}(\chi^3 d^3 N N_{\text{iter}})$

Run TEBD

Inputs:
`model`, `model_params`, N, χ, Δt, N_{iter}, `mid_steps`, `observables`, `print_to_stdout`, `evol_type`, `st_order`, `initial_state`, `rng_seed`

```
d = 2

if initial_state == "random":
    MPS = MatrixProductState(d=d, N=N, bond_dim=bond_dim, rng_seed=rng_seed)
else:
    MPS = MatrixProductState.init_from_state(initial_state)

# create Hamiltonians
if model == "ising":
    loc_ham = LocalIsingHamiltonian(N=N, **model_params)
    glob_ham = IsingHamiltonian(N=N, **model_params)
elif model == "heisenberg":
    loc_ham = LocalHeisenbergHamiltonian(N=N, **model_params)
    glob_ham = HeisenbergHamiltonian(N=N, **model_params)
else:
    raise Exception(f"Model {model} not supported")

# create TEBD object
tebd_obj = TEBD(MPS, loc_ham, glob_ham, bond_dim=bond_dim, evol_type=evol_type, st_order=st_order)

# run algorithm
observables_at_midsteps = tebd_obj.sweep(tau, num_iter, mid_steps, observables, print_to_stdout)

return observables_at_midsteps
```



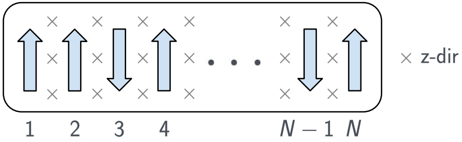
Results

TEBD on Quantum Ising Model

- TEBD can be applied to **quantum Ising model**, a quantum system composed of N spin-1/2 particles on one-dimensional lattice in presence of external magnetic field:

$$\hat{H} = J \sum_{i=1}^{N-1} \sigma_i^x \sigma_{i+1}^x + \lambda \sum_{i=1} \sigma_i^z$$

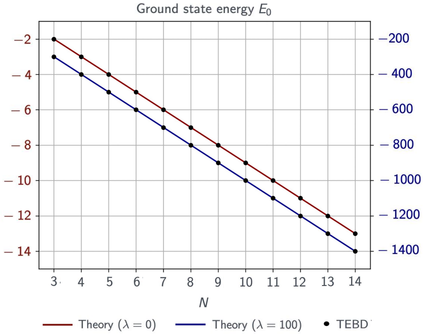
where J is coupling between neighboring spins and λ is coupling to magnetic field

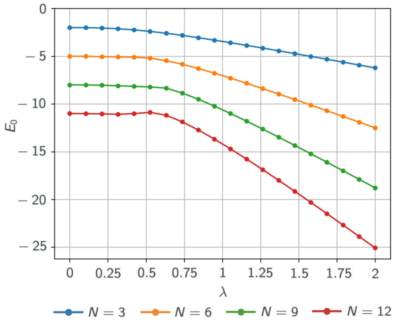


- When $\lambda = 0$, all spins align when $J < 0$ (**ferromagnetic**) and anti-align when $J > 0$ (**antiferromagnetic**). The ground state is two-fold degenerate with energy $-N+1$
- When $\lambda \to \infty$, spins align to magnetic field and ground-state energy is $-\lambda N$

Ground State Energy

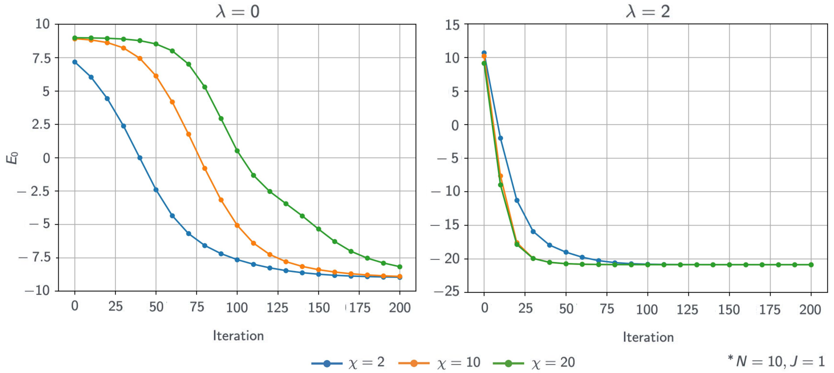
- Ground state energy E_0 can be determined by simulating system using TEBD
- In following experiments, TEBD was run for 500 iterations with a timestep of $\Delta t = 0.01$ and $\chi = 2$. The initial state was random, but was the same across all values of N. Results are robust to choice of χ
- TEBD reproduces E_0 reported in Assignment 7. Results match theoretical expectation very well: $E_0 = -N + 1$ for $\lambda = 0$ and $E_0 = -\lambda N$ for large λ





Effect of Bond Dimension

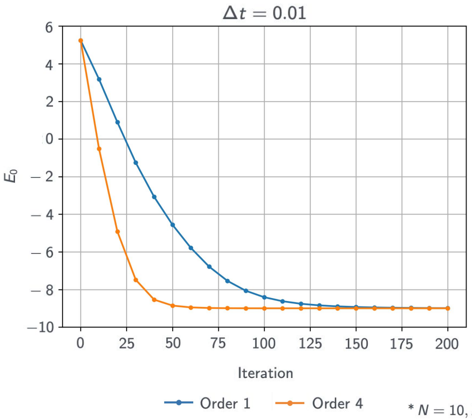
- Increasing bond dimension χ allows model to capture more entanglement, but it takes longer to converge when λ is small since long-range order is low
- When λ is beyond critical point, larger χ leads to faster convergence since degree of long-range order is higher



$^*N = 10, J = 1$

Effect of Suzuki-Trotter Order

- Increasing the order of Suzuki-Trotter decomposition improves convergence rate
- Error of first order is $\mathcal{O}(\Delta t)$ while error of fourth error is $\mathcal{O}(\Delta t^4)$



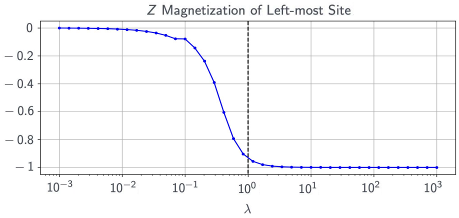
* $N = 10, \lambda = 0, J = 1$

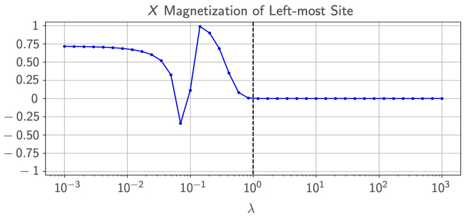
Magnetization

- Magnetization computed for site i along direction j using the magnetic operator:

$$M_{ij} = \mathbb{1}_1 \otimes \cdots \otimes \mathbb{1}_{i-1} \otimes \sigma_i^j \otimes \mathbb{1}_{i+1} \otimes \cdots \otimes \mathbb{1}_N$$

- Average magnetization is given by $\langle M_{ij} \rangle = \mathrm{Tr}(\rho M_{ij}) = \langle \Psi | M_{ij} | \Psi \rangle$
- When λ is low, magnetization along Z is zero as spins have equal probability of being $\uparrow$ or $\downarrow$. When λ increases, approaching phase transition at $\lambda = 1$, spins align to magnetic field and $\langle M_{1Z} \rangle = 0$
- When Z magnetization is -1, average X magnetization is zero as Z-polarized qubit has equal probability of being $\leftarrow$ or $\rightarrow$

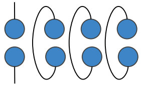




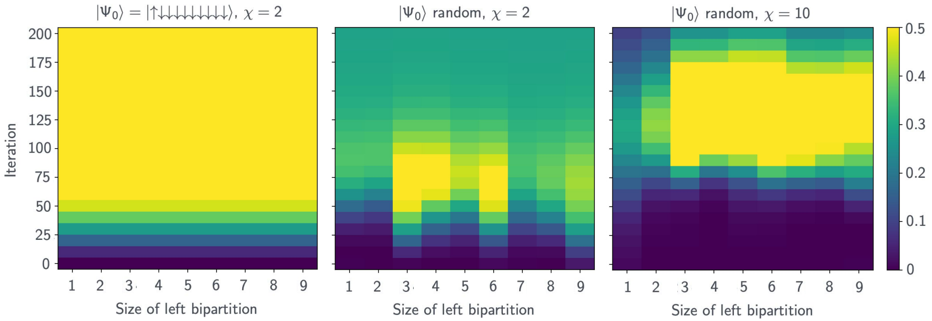
Entanglement Entropy

- Entropy of left bipartition computed by contracting density matrix tensor:

$$S = -\mathrm{Tr}(\rho_L \log \rho_L) \text{ where } \rho_L = \mathrm{Tr}_R\, \rho =$$



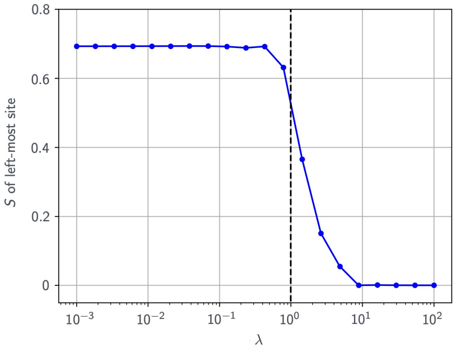
- For specific initial state $|\uparrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\rangle$, entropy starts uniform and grows uniformly
- For random state, entropy grows and spreads unevenly before becoming uniform. With bigger bond dimension χ, entropy across the chain persists for longer



Entanglement Entropy at Criticality

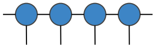
Università degli Studi di Padova

- When λ is low, entanglement entropy of left-most site is high because ground state is degenerate
- When λ exceeds critical value, entropy decreases as all spins align to magnetic field, eliminating degeneracy and creating pure reduced state

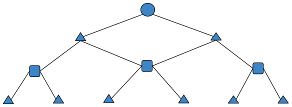


Conclusion

Conclusion



- Time-evolving block decimation is a powerful method of simulating many-body quantum systems using matrix product states
- Through efficient truncation of Hilbert space, algorithm remains computationally tractable while capturing some degree of entanglement
- TEBD can be applied to weakly-coupled one-dimensional quantum systems like Ising or Heisenberg models
- Similar techniques can be used applied using different tensor network structures, like tree tensor networks (e.g., t-MERA)



Appendix

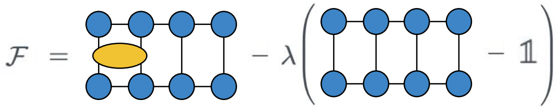
Fidelity Maximization

- TEBD is based on concept of maximizing **fidelity** of reconstruction between MPS structure and evolved state at each time t:

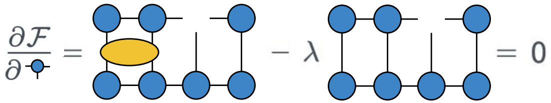
$$\mathcal{F}^* = \max_{\Psi \in M} \langle \Psi | U_t | \Psi \rangle - \lambda(\langle \Psi | \Psi \rangle - 1)$$

  where M is space of MPS tensors, and second term helps to normalize the state

- Graphically, fidelity is represented by

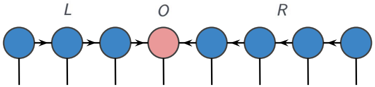


- This can be solved using **gradient descent** by differentiating $\mathcal{F}$ with respect to every tensor. For example, differentiating with respect to the third tensor leads to

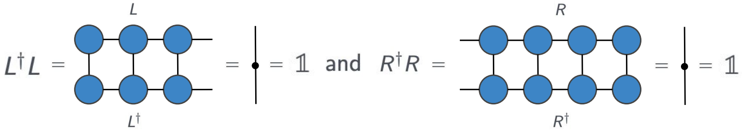


Centers of Orthogonality

- Tensor O is **center of orthogonality** if every branch connected to O is isometry between its legs and index connected to O. For example, O in the tensor network



is center of orthogonality if



- This can be done by finding the principal square roots of $L^\dagger L$ and $R^\dagger R$ and reabsorbing them into O and left and right branches, respectively
- Orthogonality centers are important in TEBD because decomposing tensor O with SVD minimizes **global** error of decomposition if O is center of orthogonality

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
- S. Montangero, "Introduction to Tensor Network Methods: Numerical Simulations of Low-Dimensional Many-Body Quantum Systems," Springer Nature, Switzerland, 2018
- G. Evenbly, "Everything you need to begin your exciting journey into the world of tensor networks," www.tensors.net
- J. Gray, "QUIMB: A Python Library for Quantum Information and Many-body Calculations," Journal of Open Source Software, vol. 3, no. 29, 2018