TEBD

David Aceituno

Contents

iTEBD on C++

The code follows the steps outlined in the following paper

> Kjäll, J. A., Zaletel, M. P., Mong, R. S. K., Bardarson, J. H. & Pollmann, F. Phase diagram of the anisotropic spin-2 XXZ model: Infinite-system density matrix renormalization group study. *Phys. Rev. B* **87,** 235106 (2013).

and the file itebd.py.

Notation

Latin letters are reserved for physical indices, and greek indices for bond indices. The order of indices in all tensors is *physical indices* → *bond indices*. In diagrammatic notation we have

$$\Lambda_{\alpha}\beta = \quad \alpha \;\boxed{\Lambda}\; \beta \quad = \quad 0 \;\boxed{\Lambda}\; 1 \tag{1}$$

$$\Gamma^{i}_{\alpha\beta} = \quad \alpha \;\overset{i}{\boxed{\Gamma}}\; \beta \quad = \quad 1 \;\overset{0}{\boxed{\Gamma}}\; 2 \tag{2}$$

$$\theta^{ij}_{\alpha\beta} = \quad \alpha \;\overset{i\ \ j}{\boxed{\theta}}\; \beta \quad = \quad 2 \;\overset{0\ \ 1}{\boxed{\theta}}\; 3 \tag{3}$$

In C++ these are explicitly objects of type `Eigen::Tensor<double,rank,Eigen::ColMajor>;`, which are `typedef`'ed into shorthand `TensorR`, where `R` is the rank (0 to 4).

Model

Hamiltonian

We study the 1D Ising model with the following two-site Hamiltonian

$$H = \begin{pmatrix} J & -g/2 & -g/2 & 0 \\ -g/2 & -J & 0 & -g/2 \\ -g/2 & 0 & -J & -g/2 \\ 0 & -g/2 & -g/2 & J \end{pmatrix}.$$

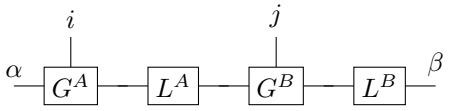
Time Evolution Operator

We now define $U = e^{-H\delta\tau}$, a unitary matrix that performs the time evolution. U is reshaped into a rank 4 tensor with dimensions (2,2,2,2), i.e. $U = U^{jk}_{j'k'}$, where j and k are the original physical indices, and j' and k' are the updated indices.

The iTEBD algorithm

Initialization

We initialize a two-site MPS state in Vidal canonical form



- Define $G^{A,i}_{\alpha\gamma} = G^{B,i}_{\delta\beta}$ with dimensions $(2,1,1)$. Set $G^{A,B}(0,0,0) = 1$ and the rest to zero. In code: `std::array<Tensor3,sites>G;`, where `constexpr int sites =2;`
- Define diagonal matrices $L^A_{\gamma\gamma} = L^B_{\delta\delta}$ with dimensions $(1,1)$. Set $L^{A,B}(0,0) = 1$. In code: `std::array<Tensor2,sites>G;`.

SVD Truncation

The SVD rank is the number of singular values obtained in the SVD decomposition. In practice the singular values in Λ are ordered in decreasingly, and the crucial observation is that their magnitude decreases exponentially. There are two reasons for wanting to truncate the smallest singular values.

- We want the maximum bond dimension to remain bounded to some upper limit. We therefore define $\chi_{\max}$, the maximum rank of each SVD decomposition simply by `long chi =15;`. If the SVD decomposition yields more than χ singular values, these are ignored.
- Since we need Λ^{-1} in the algorithm, very small singular values may harm numerical precision. Hence we define a minimum threshold for the SVD; singular values smaller than `double SVDThreshold =1e-10;` will be ignored. The constant is passed to the `Eigen::BCDSVD SVD;` object by writing `SVD.setThreshold(SVDThreshold);`.

Loop

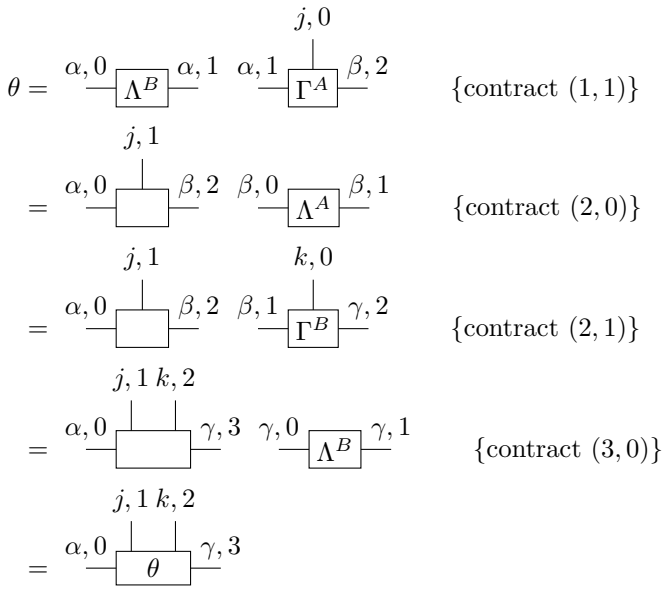
Next we have a double `for`-loop. The outer for loop steps through $N = 10000$ iterations and the inner loop iterates through both sites A and B, i.e. 2 steps.

Let's study the code line-by-line:

- [3-4] Variables $i_a, i_b = 0, 1$ if step is even and $1, 0$ if step is odd.
- [5-6] Variables χ_a, χ_b store the current dimensions so that X and Z can be reinterpreted as rank 3 tensors after the SVD. in line [20-23].
- [9-13] These lines perform eq. (25) in the article, Essentially we perform

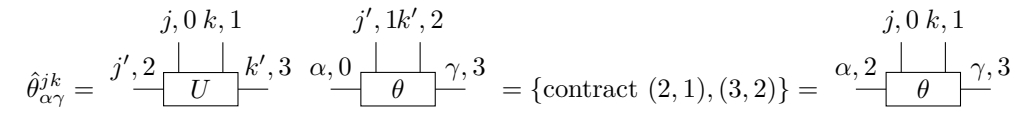
$$\theta^{jk}_{\alpha\gamma} = \sum_{\beta} \Lambda^B_\alpha \Gamma^{A,j}_{\alpha\beta} \Lambda^A_\beta \Gamma^{B,k}_{\beta\gamma} \Lambda^B_\gamma.$$

  Let's review why those particular indices are contracted in the code:



  Notice that the index order breaks our convention of "physical indices first". This is intentional. The next step fixes that automatically.

- [15] Time evolution, eq. (26) in the article. In diagrammatic representation we do



  which has the correct index order according to our convention. At the same time we flatten, or reshape the rank 4 tensor $\hat{\theta}^{jk}_{\alpha\gamma}$ down to a rank 2 tensor $\hat{\theta}_{j\alpha;k\gamma}$. To do this we must first collect the indices in order $j\alpha k\gamma$, i.e., α and k need to switch places according to the shuffling (or transpose) $(0, 2, 1, 3)$.

- [19] Eq. (27) in the article. The SVD splits theta into matrices. $\hat{\theta}_{j\alpha;k\gamma} = \sum_\beta = X_{j\alpha;\beta} Y_\beta Z_{k\gamma;\beta}$. Note that by default, Eigen does not give a transposed Z, i.e., in general we get $M = USV$ and not $M = USV^\dagger$, which is why the indices of Z are swapped compared to the article. Also, Eigen outputs the singular values Y in an array, not a square matrix.

- [21] Decide how many rows $\chi_2 < \chi_{\max}$ to keep.
- [22-25] X, Y, and Z are truncated down to χ_2 columns and then reshaped into rank 3 tensors using the dimensions χ_a, χ_b stored earlier.
- [26] Set $L^A = Y/|Y|$, i.e.set to a normalized diagonal matrix form.
- [28-30] Eq. (28a-b) in the article. We update G^A, G^B using the inverse singular values $(L^B)^{-1}$ from the previous step.

C++ code

```
for(int step = 0; step < N; step++){
  for(long i_bond = 0; i_bond < d; i_bond++) {
    long ia = mod(i_bond  ,d);
    long ib = mod(i_bond+1,d);
    long chia = G[ia].dimension(1); //index alpha (eq. 25-27)
    long chib = G[ib].dimension(2); //index gamma (eq. 25-27)

    // Construct theta matrix and do time evolution
    theta4  = L[ib]
              .contract(G[ia], idxlist1{idx2(1,1)})
              .contract(L[ia], idxlist1{idx2(2,0)})
              .contract(G[ib], idxlist1{idx2(2,1)})
              .contract(L[ib], idxlist1{idx2(3,0)});

    theta2  = U.contract(theta4, idxlist2 {idx2(2,1),idx2(3,2)})
              .shuffle(array4{0,2,1,3})
              .reshape(array2{d*chia,d*chib});

    SVD.compute(tensor2_to_matrix(theta2), Eigen::ComputeThinU | Eigen::ComputeThinV);

    chi2 = std::min(SVD.rank(),chi);
    X = matrix_to_tensor3(SVD.matrixU().leftCols(chi2),{d,chia,chi2});
    Z = matrix_to_tensor3(SVD.matrixV().leftCols(chi2),{d,chib,chi2})
                                          .shuffle(array3{0,2,1});
    Y = matrix_to_tensor1(SVD.singularValues().head(chi2));
    L[ia] = asDiagonal(Y / SVD.singularValues().head(chi2).norm());

    G[ia] = inverseDiagonal(L[ib]).contract(X, idxlist1{idx2(0,1)})
                                  .shuffle(array3{1,0,2});
    G[ib] = Z.contract(inverseDiagonal(L[ib]), idxlist1{idx2(2,0)} );
  }
}
```

Energy calculation

C++ code

```
    Tensor4 GG;
    Tensor4 sGG;
    Tensor4 C;
    Array2d E;

    for(long i_bond = 0; i_bond < d; i_bond++) {
        long ia = mod(i_bond  ,d);
        long ib = mod(i_bond+1,d);
        A   = G[ia].contract(L[ia], idxlist1{idx2(2,0)});
        B   = G[ia].contract(L[ib], idxlist1{idx2(2,0)});
        GG  = B.contract(A, idxlist1{idx2(2,1)});
        sGG = L[ib].contract(GG, idxlist1{idx2(0,1)}).shuffle(array4{1,2,0,3});
        C   = sGG.contract(H4,idxlist2{idx2(0,2), idx2(1,3)}).shuffle(array4{2,3,0,1});
        Tensor0 result =
            sGG.conjugate().contract(C,idxlist4{idx2(0,0),idx2(1,1),idx2(2,2),idx2(3,3)}).eval();
        E(i_bond) = result(0);
    }
```