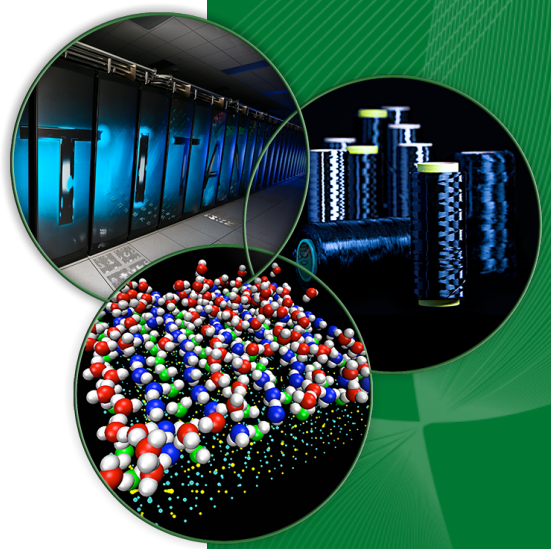
DMRG Algorithm

October 5, 2016



Let us define *block* to mean a finite set of sites. Let $N > 0$ be called the total number of sites. Let C denote the states of a single site. This set is model dependent. For the Hubbard model it is given by: $C = \{e, \uparrow, \downarrow, (\uparrow, \downarrow)\}$, where e is a formal element that denotes an empty state. For the t-J model it is given by $C = \{e, \uparrow, \downarrow\}$, and for the spin 1/2 Heisenberg model by $C = \{\uparrow, \downarrow\}$, for the Hubbard model with 2 orbitals, C is composed of 16 elements. A *real-space-based Hilbert space* $\mathcal{V}$ on a block B and set C is a Hilbert space with basis C^B. I will simply denote this as $\mathcal{V}(B)$ and assume that C is implicit and fixed. This is a Hilbert space because (i) it is a vector space over $\mathbb{C}$, and (ii) it has a norm.[1] It is a vector space: The elements $\psi \in C^B$ are endowed[2] of a formal sum and multiplication. It has a norm: Because it has an an inner product: for every $\psi, \psi' \in C^B$ as $(\psi, \psi') = 0$ if ψ and ψ' are different and 1 if equal.[3]

[1] Or it has the topology induced by the norm, making the space separable also.

[2] Elements of the Hilbert space are usually noted with the so-called Dirac notation as $|\psi\rangle$.

[3] The inner product (ψ, ψ') is usually noted in the so-called Dirac notation as $\langle\psi|\psi'\rangle$.

Notation

A *real-space-based Hilbert space* can also be thought of as the external product space of $\#C$ Hilbert spaces on a site, one for each site in block B. The basis $\psi \in C^B$ of $\mathcal{V}$ is the so-called *computational (or natural) basis* of $\mathcal{V}$. $C^B = \bigotimes_B C$, so that $\psi \in C^B$ can be written as $\psi_{i_0} \otimes \psi_{i_1} \otimes \cdots \otimes \psi_{i_{N-1}}$ where ψ_{i_n} is in the $n-$th space B, and it is the i_n-th element of that space.
I give a procedural description of the DMRG algorithm[1, 2] in the following. We start with an initial block S (the initial system) and E (the initial environment). Consider two sets of blocks X and Y. We will be adding blocks from X to S, one at a time, and from Y to E, one at a time. Again, note that X and Y are sets of blocks whereas S and E are blocks. Now we start a loop for the DMRG "infinite" algorithm by setting $step = 0$ and $\mathcal{V}_R(S) \equiv \mathcal{V}(S)$ and $\mathcal{V}_R(E) \equiv \mathcal{V}(E)$. The system is grown by adding the sites in X_{step} to it, and let $S' = S \cup X_{step}$, i.e. the $step$-th block of X to S is added to form the block S'; likewise, let $E' = E \cup Y_{step}$. Let us form the following product Hilbert spaces: $\mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$ and $\mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$ and their union $\mathcal{V}(S') \otimes \mathcal{V}(E')$ which is disjoint.

Notation

We will consider now operators acting on $\mathcal{V}$. A special operator is called the Hamiltonian. Consider $\hat{H}_{S'\cup E'}$, the Hamiltonian operator, acting on $\mathcal{V}(S')\otimes\mathcal{V}(E')$. We diagonalize $\hat{H}_{S'\cup E'}$ (using Lanczos) to obtain its lowest eigenvector:

$$|\psi\rangle = \sum_{\alpha\in\mathcal{V}(S'),\beta\in\mathcal{V}(E')} \psi_{\alpha,\beta}|\alpha\rangle\otimes|\beta\rangle, \tag{1}$$

where $\{|\alpha\rangle\}$ is a basis of $\mathcal{V}(S')$ and $\{|\beta\rangle\}$ is a basis of $\mathcal{V}(E')$.
Let us define the density matrices for system:

$$(\hat{\rho}_S)_{\alpha,\alpha'} = \sum_{\beta\in\mathcal{V}(E')} \psi^*_{\alpha',\beta}\psi_{\alpha,\beta} \tag{2}$$

in $\mathcal{V}(S')$, and environment:

$$(\hat{\rho}_E)_{\beta,\beta'} = \sum_{\alpha\in\mathcal{V}(S')} \psi^*_{\alpha,\beta'}\psi_{\alpha,\beta} \tag{3}$$

in $\mathcal{V}(E')$.

Notation

We then diagonalize $\hat{\rho}_S$, and obtain its eigenvalues and eigenvectors, $W^S_{\alpha,\alpha'}$ in $\mathcal{V}(S')$ ordered in decreasing eigenvalue order.
We change basis for the operator $H^{S'}$ (and other operators as necessary), as follows:

$$(H^{S'\text{new basis}})_{\alpha,\alpha'} = (W^S)^{-1}_{\alpha,\gamma}(H^{S'})_{\gamma,\gamma'}W^S_{\gamma',\alpha'}. \tag{4}$$

We proceed in the same way for the environment, diagonalize $\hat{\rho}_E$ to obtain ordered eigenvectors W^E, and define $(H^{E'\text{new basis}})_{\alpha,\alpha'}$.
Let m_S be a fixed number that corresponds to the number of states in $\mathcal{V}(S')$ that we want to keep. Consider the first m_S eigenvectors W^S, and let us call the Hilbert space spanned by them, $\mathcal{V}_R(S')$, the DMRG-reduced Hilbert space on block S'. If $m_S \geq \#\mathcal{V}(S')$ then we keep all eigenvectors and there is effectively no truncation. We truncate the matrices $(H^{S'\text{new basis}})$ (and other operators as necessary) such that they now act on this truncated Hilbert space, $\mathcal{V}_R(S')$. We proceed in the same manner for the environment.

Notation

Now we increase *step* by 1, set $S \leftarrow S'$, $\mathcal{V}_R(S) \leftarrow \mathcal{V}_R(S')$, $H_{S'} \leftarrow H_S$, and similarly for the environment, and continue with the growth phase of the algorithm.
In the infinite algorithm, the number of sites in the system and environment grows as more steps are performed. After this infinite algorithm, a finite algorithm is applied where the environment is shrunk at the expense of the system, and the system is grown at the expense of the environment. During the finite algorithm phase the total number of sites remains constant allowing for a formulation of DMRG as a variational method in a basis of matrix product states.

Why the DMRG works?

The advantage of the DMRG algorithm is that the truncation procedure described above keeps the error bounded and small. Assume $m_S = m_E = m$. At each DMRG step[3] the truncation error $\epsilon_{tr} = \sum_{i>m} \lambda_i$, where λ_i are the eigenvalues of the truncated density matrix ρ_S in decreasing order. The parameter m should be chosen such that ϵ_{tr} remains small, say [3] $\epsilon_{tr} < 10^{-6}$. For critical 1D systems ϵ_{tr} decays as a function of m with a power law, while for 1D system away from criticality it decays exponentially. For a more detailed description of the error introduced by the DMRG truncation in other systems see [3, 4, 5, 6].

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E).$

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E).$
2.

$$S' = S \cup X_{step}, \mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$$
$$E' = E \cup Y_{step}, \mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$$

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E).$
2. $$S' = S \cup X_{step}, \mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$$
   $$E' = E \cup Y_{step}, \mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$$
3. Diagonalize $\hat{H}_{S' \cup E'}$ over $\mathcal{V}(S') \otimes \mathcal{V}(E')$ to obtain ψ

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E).$
2. $$S' = S \cup X_{step}, \mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$$
   $$E' = E \cup Y_{step}, \mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$$
3. Diagonalize $\hat{H}_{S' \cup E'}$ over $\mathcal{V}(S') \otimes \mathcal{V}(E')$ to obtain ψ
4. Obtain Density Matrix $\hat{\rho}_S$ from ψ

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E).$
2. $$S' = S \cup X_{step}, \mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$$
   $$E' = E \cup Y_{step}, \mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$$
3. Diagonalize $\hat{H}_{S' \cup E'}$ over $\mathcal{V}(S') \otimes \mathcal{V}(E')$ to obtain ψ
4. Obtain Density Matrix $\hat{\rho}_S$ from ψ
5. Diagonalize Density Matrix $\hat{\rho}_S$ to obtain $W^S_{\alpha,\alpha'}$ (similar for E)

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E).$
2. $$S' = S \cup X_{step}, \mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$$
   $$E' = E \cup Y_{step}, \mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$$
3. Diagonalize $\hat{H}_{S' \cup E'}$ over $\mathcal{V}(S') \otimes \mathcal{V}(E')$ to obtain ψ
4. Obtain Density Matrix $\hat{\rho}_S$ from ψ
5. Diagonalize Density Matrix $\hat{\rho}_S$ to obtain $W^S_{\alpha,\alpha'}$ (similar for E)
6. Truncate W^S if necessary (similar for W^E)

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E).$
2. $$S' = S \cup X_{step}, \mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$$
   $$E' = E \cup Y_{step}, \mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$$
3. Diagonalize $\hat{H}_{S' \cup E'}$ over $\mathcal{V}(S') \otimes \mathcal{V}(E')$ to obtain ψ
4. Obtain Density Matrix $\hat{\rho}_S$ from ψ
5. Diagonalize Density Matrix $\hat{\rho}_S$ to obtain $W^S_{\alpha,\alpha'}$ (similar for E)
6. Truncate W^S if necessary (similar for W^E)
7. $H^{S'}$ into $W^{S\dagger} H^{S'} W^S$ (similar for E)

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E).$
2. $$S' = S \cup X_{step}, \mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$$
   $$E' = E \cup Y_{step}, \mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$$
3. Diagonalize $\hat{H}_{S' \cup E'}$ over $\mathcal{V}(S') \otimes \mathcal{V}(E')$ to obtain ψ
4. Obtain Density Matrix $\hat{\rho}_S$ from ψ
5. Diagonalize Density Matrix $\hat{\rho}_S$ to obtain $W^S_{\alpha,\alpha'}$ (similar for E)
6. Truncate W^S if necessary (similar for W^E)
7. $H^{S'}$ into $W^{S\dagger} H^{S'} W^S$ (similar for E)
8. Auxiliary matrices $c^{S'}_\gamma$ into $W^{S\dagger} c^{S'}_\gamma W^S$ (similar for E)

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E).$
2. $$S' = S \cup X_{step}, \mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$$
   $$E' = E \cup Y_{step}, \mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$$
3. Diagonalize $\hat{H}_{S' \cup E'}$ over $\mathcal{V}(S') \otimes \mathcal{V}(E')$ to obtain ψ
4. Obtain Density Matrix $\hat{\rho}_S$ from ψ
5. Diagonalize Density Matrix $\hat{\rho}_S$ to obtain $W^S_{\alpha,\alpha'}$ (similar for E)
6. Truncate W^S if necessary (similar for W^E)
7. $H^{S'}$ into $W^{S\dagger} H^{S'} W^S$ (similar for E)
8. Auxiliary matrices $c^{S'}_\gamma$ into $W^{S\dagger} c^{S'}_\gamma W^S$ (similar for E)
9. $step$++, $S \leftarrow S'$, $\mathcal{V}_R(S) \leftarrow \mathcal{V}_R(S')$, $H_{S'} \leftarrow H_S$ (similar for E)

DMRG step by step

1. $step = 0, \mathcal{V}_R(S) \equiv \mathcal{V}(S), \mathcal{V}_R(E) \equiv \mathcal{V}(E)$.
2. $$S' = S \cup X_{step}, \mathcal{V}(S') = \mathcal{V}_R(S) \otimes \mathcal{V}(X_{step})$$
   $$E' = E \cup Y_{step}, \mathcal{V}(E') = \mathcal{V}_R(E) \otimes \mathcal{V}(Y_{step})$$
3. Diagonalize $\hat{H}_{S' \cup E'}$ over $\mathcal{V}(S') \otimes \mathcal{V}(E')$ to obtain ψ
4. Obtain Density Matrix $\hat{\rho}_S$ from ψ
5. Diagonalize Density Matrix $\hat{\rho}_S$ to obtain $W^S_{\alpha,\alpha'}$ (similar for E)
6. Truncate W^S if necessary (similar for W^E)
7. $H^{S'}$ into $W^{S\dagger} H^{S'} W^S$ (similar for E)
8. Auxiliary matrices $c^{S'}_\gamma$ into $W^{S\dagger} c^{S'}_\gamma W^S$ (similar for E)
9. $step$++, $S \leftarrow S'$, $\mathcal{V}_R(S) \leftarrow \mathcal{V}_R(S')$, $H_{S'} \leftarrow H_S$ (similar for E)
10. Goto 2

See 2. Implementation details in 7.

Hamiltonian Construction

The left block linear or Hilbert space has size n_l, and the right block n_r. The left block plus right block space, the so-called superblock linear space, is the *outer* or *Kronecker* product of left and right spaces, and has size $n_l \times n_r$.

$$H' = H_L \otimes I_R + I_L \otimes H_R + \sum_{\gamma=0}^{\gamma<\Gamma} c_L^{\gamma} \otimes c_R^{\gamma},$$

where H'_L is a $n_l \times n_l$ CRS matrix, H'_R a $n_l \times n_r$ CRS matrix, $c_L'^{\gamma}$ are Γ CRS matrices of rank n_l, and $c_R'^{\gamma}$ are Γ CRS matrices of rank n_r.

Hamiltonian With Symmetries

The square matrix H' of rank $n_l \times n_r$, when written in an appropriate basis, is block diagonal. Only one of those blocks need to be diagonalized, the needed block is usually known through the use of quantum numbers. A more accurate expression of H' is then

$$H = P_{SE}^{-1} \left(H_L \otimes I_R + I_L \otimes H_R + \sum_{\gamma=0}^{\gamma<\Gamma} c_L^{\gamma} \otimes c_R^{\gamma} \right) P_{SE}, \tag{5}$$

where P_{SE} is a permutation of indices in the superblock basis.
For a fully stored approach we need to compute the matrix block for H. *show where this is done in the code*
For a fully on-the-fly approach (normally used) we need to compute the vector x given a vector y, where $x = Hy$. *show where this is done in the code*

Is there more structure to the P_{SE} permutation?

States in S each has a quantum number (non-negative integer) q_i, i a state in $\mathcal{V}(\mathcal{S})$,[4] q_i are ordered in increasing number. States in E each has a quantum number (non-negative integer) q_j, j a state in E, q_j are ordered in increasing number. We construct the space in `SE = space in S cartesian product with space in E`. In other words (i, j) is a state in SE iff i is a state in S, and j a state in E. We assign the quantum number $q_{(i,j)} = q_i + q_j$ to state (i, j) in SE. By construction, you can see that $q_{(i,j)}$ is not (necessarily) ordered in increasing number. We define the permutation P_{SE} that orders $q_{(i,j)}$.

[4] Let me call the Hilbert space $\mathcal{V}(\mathcal{S})$ just S only for the purposes of this answer.

How big is a single diagonal block of H matrix compared full H' matrix

This is model dependent. *If there is truncation what follows will be bounds, not actual sizes.* Let's say we have N sites. For the Hubbard model, blocks are labeled by a pair $(N_\uparrow, N_\downarrow)$ such that $0 \leq N_\uparrow \leq N$ (same for $N_\downarrow$). There are then $(N+1)^2$ blocks I think. Each block has size $C(N, N_\uparrow)C(N, N_\downarrow)$ The block we are usually interested in is the largest, the one with $N_\uparrow = N_\downarrow = N/2$.

For the Heisenberg model, blocks are labeled by $N_\uparrow$ such that $0 \leq N_\uparrow \leq N$. There are then $(N+1)$ blocks I think. Each block has size $C(N, N_\uparrow)$. The block we are usually interested in is the largest, the one with $N_\uparrow = N/2$.

For the t-J model, blocks are labeled by $(N_\uparrow, N_\downarrow)$ such that $0 \leq N_\uparrow \leq N$, $0 \leq N_\downarrow \leq N$, and $N_\uparrow + N_\downarrow \leq N$. There are then FIXME blocks. Each block has size $C(N, N_\uparrow)C(N - N_\uparrow, N_\downarrow)$. The block we are usually interested in is the largest, the one with $N_\uparrow = N_\downarrow = N/4$. Etc, for other models.

Can you comment on the sizes and sparseness of H_L, H_R, C_L, C_R and whether there are further structure or pattern in these matrices?

I'll be using L and S as synonyms, same for R and E [5] Anyway, States in L each has a quantum number (non-negative integer) q_i, $i \in L$, q_i are ordered in increasing number. States in R each has a quantum number (non-negative integer) q_j, $j \in R$, q_j are ordered in increasing number. H_L is a matrix in space L, with the property that $H_L(i1, i2) = 0$ if $q_{i1} \neq q_{i2}$. That is, H_L is said to respect symmetry. H_R is a matrix in space R, with the property that $H_R(j1, j2) = 0$ if $q_{j1} \neq q_{j2}$. That is, H_R also respects symmetry.
C_L and C_R are model dependent, and in general do not respect symmetry. Different Cs might have different symmetries also.

[5] But please don't confuse sites with states in spaces constructed in or out of a block of sites. I don't think I have to mention blocks of sites at all, so all these letters can be used to refer to the vector spaces.

Can you comment on...? (continued)

For the Hubbard model, Cs are destruction operators. Therefore, $C_L(i1, i2) = 0$ unless $q_{i1} \neq f_C(q_{i2})$ where $f_C : \{0, 1, ...\} \rightarrow \{0, 1, ...\}$ is the function associated with the symmetry of C.

For the Heisenberg model, there are two kinds of Cs: spin down-flip (let's call them C1) and Sz operators (let's call them C2). $C1_L(i1, i2) = 0$ if $q_{i1} \neq f_C1(q_i2)$ where $f_{C1} : \{0, 1, ...\} \rightarrow \{0, 1, ...\}$ is the function associated with the symmetry of C1. $C2_L(i1, i2) = 0$ if $q_{i1} \neq q_{i2}$, that is, C2 respects symmetry.

For the Heisenberg model, there are 3 kinds of Cs, let's call them C1 and C2 and C3. Therefore, $CX_L(i1, i2) = 0$ if $q_{i1} \neq f_CX(q_{i2})$ where $f_{CX} : \{0, 1, ...\} \rightarrow \{0, 1, ...\}$ is the function associated with the symmetry of CX, for X=1,2,3. Also, f_{C3} is the identity, because C3 is the density operator, which (like the Hamiltonian) respects quantum numbers.

I guess the model dependence of symmetries can be encoded in the f_C functions.

All steps at a glance

```
// In file DmrgSolver.h

// set initial matrices
// see next slide

// Growth phase == INFINITE LOOP PHASE
// Note: Nothing is infinite here, it's just a name
infiniteDmrgLoop(S,X,Y,E,pS,pE,*psi);

// Convergence phase == FINITE LOOP PHASE
finiteDmrgLoops(S,E,pS,pE,*psi);
```

WARNING

Code is taken from DMRG++ but very simplified. Indentation has been modified to fit the slides. Applies to all slides that follow.

Set initial Matrices

```
// In file DmrgSolver.h
VectorOperatorType creationMatrix;
SparseMatrixType hmatrix;
SymmetryElectronsSzType q;

model_.setNaturalBasis(creationMatrix,
                                hmatrix,q,E,0.0);
pE.setVarious(E,hmatrix,q,creationMatrix);

model_.setNaturalBasis(creationMatrix,
                  hmatrix,q,S,time);
pS.setVarious(S,hmatrix,q,creationMatrix);
```

pS (system)

pE (environ)

Growth Phase (a.k.a. Infinite Loop Phase)

```
void infiniteDmrgLoop(...) { // In DmrgSolver.h
  lrs_.left(pS); lrs_.right(pE); checkpoint_.push(pS,pE);

  for (SizeType step=0;step<X.size();step++) {
    // grow system
    lrs_.growLeftBlock(model_,pS,X[step],time);
    // grow environment
    lrs_.growRightBlock(model_,pE,Y[step],time);

    // super = left cross right
    updateQuantumSector(lrs_.sites(),INFINITE,step);
    lrs_.setToProduct(quantumSector_);

    // continues on next slide...
```

Growth Phase (continued)

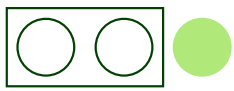
```
    // ...continued from previous slide
    // diagonalize H:
    energy_ = diagonalization_(psi,INFINITE,X[step],ystep);

    // construct rho, diag rho, truncate, change basis:
    truncate_.changeBasis(pS,pE,psi,m);

    checkpoint_.push(pS,pE); // save for shrinking later
  }
}
```

"Infinite" DMRG

system

environment

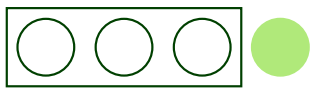
"Infinite" DMRG



system

environment

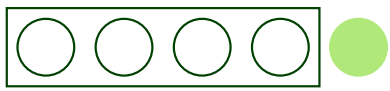
"Infinite" DMRG



system

environment

"Infinite" DMRG



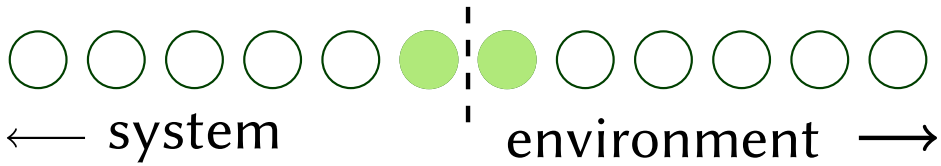
system

environment

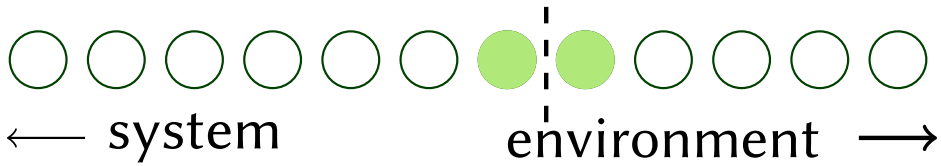
"Infinite" DMRG

system environment

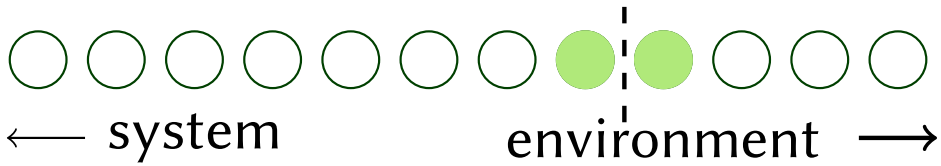
"Finite" DMRG



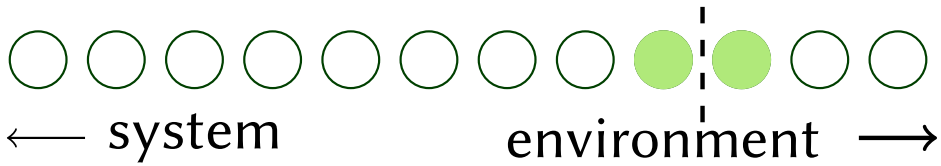
"Finite" DMRG



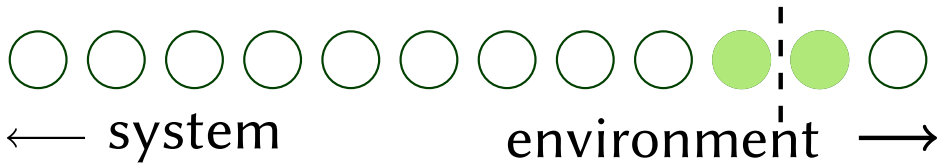
"Finite" DMRG



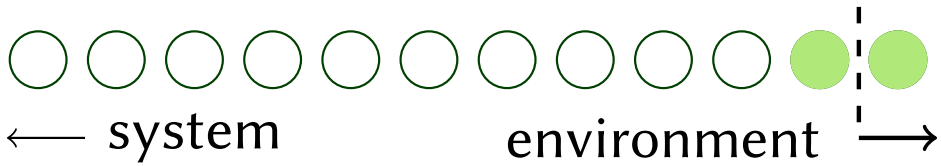
"Finite" DMRG



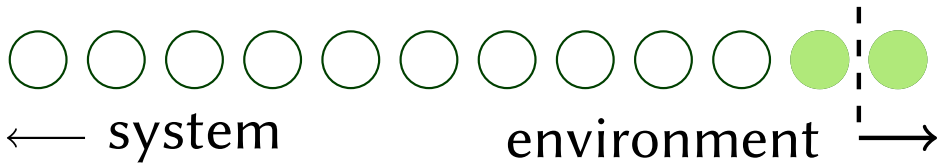
"Finite" DMRG



"Finite" DMRG

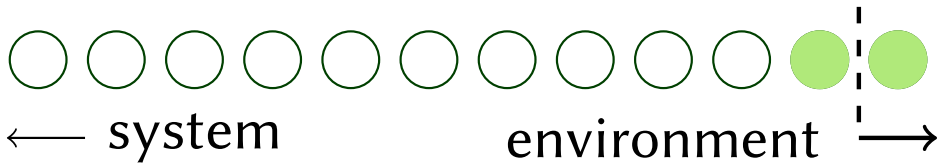


"Finite" DMRG



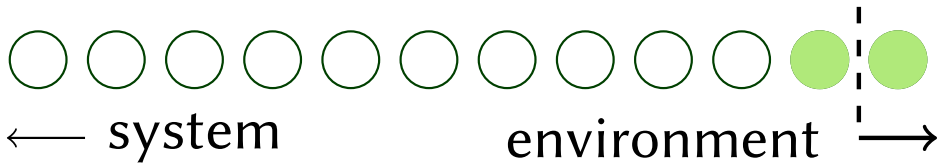
How do we shrink a basis?

"Finite" DMRG



How do we shrink a basis? Answer: We just use a previously saved one.

"Finite" DMRG



How do we shrink a basis? Answer: We just use a previously saved one.

Each vertical dashed line defines a center of orthogonality, and defines a Hilbert space basis, referred to as the DMRG basis at that center.

Convergence Phase (a.k.a. Finite Loop Phase)

```
void finiteDmrgLoops(...) { // In DmrgSolver.h
  for (SizeType i=0;i<parameters_.finiteLoop.size();i++)  {
    finiteStep(S,E,pS,pE,i,psi);
  }
}
```

Convergence Phase (a.k.a. Finite Loop Phase)

```
void finiteStep(...) { // In DmrgSolver.h
  while (true) {
    if (direction==EXPAND_SYSTEM) {
       lrs_.growLeftBlock(model_,pS,...);
       lrs_.right(checkpoint_.shrink(ENVIRON,target));

    } else {
       lrs_.growRightBlock(model_,pE,...);
       lrs_.left(checkpoint_.shrink(SYSTEM,target));
    }

    updateQuantumSector(lrs_.sites(),direction,stepCurrent_);
    lrs_.setToProduct(quantumSector_);

    diagonalization_(target,direction,...); // diag H

    // construct rho, diag rho, truncate, change basis:
    changeTruncateAndSerialize(pS,pE,target,keptStates,...);
}
```

Basis Class (Symmetries)

```
template<...>
class Basis { // In Basis.h
...
private:

  VectorSizeType quantumNumbers_;
  VectorSizeType electrons_;
  VectorSizeType partition_;
  VectorSizeType permutationVector_;

  // sites over which this basis is built
  BlockType block_;
 };
```

Permutation and Partition Vectors

ADD HERE QN EXPLANATION, EMAIL BY ED
The quantum numbers need to be reordered such that they are in increasing order, leading to a permutation.
The partition of the basis labels where the quantum number changes. Let us say that the quantum numbers of the reordered basis states are

$$\{3, 3, 3, 3, 8, 8, 9, 9, 9, 15, \cdots\}.$$

Then we define a vector named "partition", such that partition[0]=0, partition[1]=4, because the quantum number changes in the 4th position (from 3 to 8), and then partition[2]=6, because the quantum number changes again (from 8 to 9) in the 6th position, etc. Now we know that our Hamiltonian matrix will be composed first of a block of 4×4, then of a block of 2×2, etc.

BasisWithOperators Class

```
template<typename OperatorsType_> // In BasisWithOperators.h
class BasisWithOperators : public OperatorsType_::BasisType {
...
private:

  OperatorsType operators_;
  VectorSizeType operatorsPerSite_;
};
```

```
template<typename BasisType_>
class Operators { // In Operators.h
...
private:

  // OperatorType is of type Operator in Operator.h
  std::vector<OperatorType> operators_;
  SparseMatrixType hamiltonian_;
};
```

LeftRightSuper Class

```
template<typename BasisWithOperatorsType_,
         typename SuperBlockType>
class LeftRightSuper { // In LeftRightSuper.h
...
private:

        BasisWithOperatorsType* left_;
        BasisWithOperatorsType* right_;
        SuperBlockType* super_;
};
```

Here explain left of the superblock, and left of the left, and left of the right, etc. ADD FIGURE

LeftRightSuper Class

```
class LeftRightSuper { // In LeftRightSuper.h
public:

  void growLeftBlock(...) {
    grow(*left_,model,pS,X,GROW_TO_THE_RIGHT,time);
  }

  void growRightBlock(...) {
    grow(*right_,model,pE,X,GROW_TO_THE_LEFT,time);
  }
}; // class LeftRightSuper
```

Growth (In Words)

Local operators are set for the basis in question with a call to BasisWithOperators's member function setVarious. When adding sites to the system or environment the program does a full outer product, i.e., it increases the size of all local operators. This is performed by the call to setToProduct in the grow function, which actually calls leftOrRight.setToProduct. This function also recalculates the Hamiltonian in the outer product of (i) the previous system basis *pS*, and (ii) the basis *Xbasis* corresponding to the site(s) that is (are) being added. To do this, the Hamiltonian connection between the two parts needs to be calculated and added, and this is done in the call to addHamiltonianConnection. Finally, the resulting BasisWithOperators object for the outer product, leftOrRight, is set to contain this full Hamiltonian with the call to leftOrRight.setHamiltonian(matrix).

Growth (In LeftRightSuper.h)

```
void grow(...) {// add X to pS and put result in left_
    // [omitted] set hmatrix, q, and creationMatrix;

    BasisWithOperatorsType Xbasis("Xbasis");
    Xbasis.setVarious(X,hmatrix,q,creationMatrix);

    leftOrRight.setToProduct(pS,Xbasis,dir);
    SparseMatrixType matrix=leftOrRight.hamiltonian();
    ThisType* lrs;
    BasisType* leftOrRightL =  &leftOrRight;
    if (dir==GROW_TO_THE_RIGHT)
      lrs = new ThisType(pS,Xbasis,*leftOrRightL);
    else
      lrs = new  ThisType(Xbasis,pS,*leftOrRightL);

    model.addHamiltonianConnection(matrix,*lrs,time);
    delete lrs; leftOrRight.setHamiltonian(matrix);
}
```

Update Quantum Sector

```
void updateQuantumSector(...) {
  quantumSector_ =
   SymmetryElectronsSzType::getQuantumSector(...);
}

// set a vector of properties for symmetry sector
// to be calculated, the symmetry sector of interest
setTargetNumbers(v,...);
// pack the vector into a single number and return it
return getQuantumSector(v,...);
```

Set To (Kronecker) Product

```
// super = left cross right
// do everything for class Basis but
// do not build operators
// This call is in DmrgSolver.h
lrs_.setToProduct(quantumSector_);

// Function is implemented in Basis.h and
// BasisWithOperators.h
// set this basis to the outer product of
// basis2 and basis3 or basis3 and basis2
// depending on dir
void setToProduct(... basis2,... basis3,int dir) {
 if (dir==GROW_RIGHT) setToProduct(basis2,basis3);
 else setToProduct(basis3,basis2);
}
```

Set To (Kronecker) Product, in BasisWithOperators.h

```
void setToProduct(... basis2,... basis3) {
 // reorder the basis
 this->setToProduct(basis2,basis3);

 // deal with operators
 SizeType x = basis2.numberOfOperators()
              +basis3.numberOfOperators();

 for (SizeType i=0;i<this->numberOfOperators();i++) {
  if (i<basis2.numberOfOperators()) {
    const OperatorType& myOp = basis2.getOperatorByIndex(i);
    operators_.externalProduct(i,myOp,basis3.size(),...);

  } else {
    const OperatorType& myOp = basis3.getOperatorByIndex
            (i-basis2.numberOfOperators());
    operators_.externalProduct(i,myOp,basis2.size(),...);
  } // continued in next slide
```

Set To (Kronecker) Product (continued)

```
    // Calc. hamiltonian
    operators_.outerProductHamiltonian(...);

    // re-order operators and hamiltonian
    operators_.reorder(this->permutationVector());

    // update operators per site [omitted]
}
```

Set To (Kronecker) Product (continued)

The quantum numbers of the original (untransformed) real-space basis are set by the model class, whereas the quantum numbers of outer products are handled by the class Basis, function setToProduct. This can be done because if $|a\rangle$ has quantum number q_a and $|b\rangle$ has quantum number q_b, then $|a\rangle \otimes |b\rangle$ has quantum number $q_a + q_b$. Basis knows how quantum numbers change when we change the basis: they do not change since the DMRG transformation preserves quantum numbers; and Basis also knows what happens to quantum numbers when we truncate the basis: quantum numbers of discarded states are discarded. In this way, symmetries are implemented efficiently, with minimal dependencies and in a model-independent way.

Set To (Kronecker) Product (continued)

```
void setToProduct(...su2Symmetry2,...su2Symmetry3) {
  block_.clear();
  utils::blockUnion(block_,
    su2Symmetry2.block_,su2Symmetry3.block_);

  SizeType ns = su2Symmetry2.size();
  SizeType ne = su2Symmetry3.size();
  quantumNumbers_.clear(); electrons_.clear();
  for (SizeType j=0;j<ne;j++) for (SizeType i=0;i<ns;i++) {
        quantumNumbers_.push_back(
          su2Symmetry2.quantumNumbers_[i]+
          su2Symmetry3.quantumNumbers_[j]);
              electrons_.push_back(su2Symmetry2.electrons(i)+
                                   su2Symmetry3.electrons(j));
  }}

  // order quantum numbers of combined basis:
  findPermutationAndPartition(); reorder();
}
```

Diagonalization of H

```
RealType operator()(...) {
  assert(direction == WaveFunctionTransfType::INFINITE);
  RealType gsEnergy = internalMain_(...);
  return gsEnergy;
}

RealType operator()(...) {
  assert(direction != WaveFunctionTransfType::INFINITE);
  RealType gsEnergy = internalMain_(...);
  return gsEnergy;
}
```

Diagonalization of H (continued)

```
void internalMain_(...) {
  VectorComplexOrRealType vecSaved; VectorRealType energySaved;
  SizeType total = lrs.super().partition()-1;
  energySaved.resize(total); vecSaved.resize(total);
  VectorSizeType weights(total);

  for (SizeType i=0;i<total;i++) {
    SizeType bs = lrs.super().partition(i+1)-
                  lrs.super().partition(i);
    weights[i]=bs;
    // Do only one symmetry sector
    SizeType qn = ...;
    if (qn != quantumSector_ && !findSymmetrySector)
              weights[i]=0;

        vecSaved[i].resize(weights[i]);
   } // continued on next slide
```

Diagonalization of H (continued)

```
    // ...continued from previous slide
    // initial guess is important
    VectorWithOffsetType initialVector(weights,lrs.super());
    target.initialGuess(initialVector,block);

    for (SizeType i=0;i<total;i++) {
      if (weights[i]==0) continue;
      TargetVectorType initialVectorBySector(weights[i]);
      initialVector.extract(initialVectorBySector,i);
      diagonaliseOneBlock(i,vecSaved[i],gsEnergy,...);
      energySaved[i]=gsEnergy;
    }

    return gsEnergy;
}
```

Diagonalization of H (continued)

```
void diagonaliseOneBlock(...) {
  PsimagLite::String options = parameters_.options;
  typename ModelType::ModelHelperType
                  modelHelper(i,lrs,targetTime,threadId);

  if (options.find("debugmatrix")!=PsimagLite::String::npos) {
    SparseMatrixType fullm;
    model_.fullHamiltonian(fullm,modelHelper);
    return;
  }

  diagonaliseOneBlock(...); // second function with same name
}
```

Diagonalization of H (continued)

```
void diagonaliseOneBlock(...) { // function overload (2)
  typename LanczosOrDavidsonBaseType::MatrixType
  lanczosHelper(&model_,&modelHelper,rs);
  ParametersForSolverType params(io_,"Lanczos");
  LanczosOrDavidsonBaseType* lanczosOrDavidson = 0;

  bool useDavidson = (options.find("useDavidson") != npos);
  if (useDavidson) lanczosOrDavidson = new
        DavidsonSolverType(lanczosHelper,params);
  else lanczosOrDavidson = new
        LanczosSolverType(lanczosHelper,params);

  tmpVec.resize(lanczosHelper.rank());
  energyTmp =computeLevel(*lanczosOrDavidson,...);
  if (lanczosOrDavidson) delete lanczosOrDavidson;
}
```

Diagonalization of H (continued)

computeLevel in Diagonalization.h calls lanczosOrDavidson->computeExcitedState(...) which is in PsimagLite/src/LanczosSolver.h and does the Lanczos algorithm.
It uses lanczosHelper as the matrix object. lanczosHelper is of class MatrixVectorOnTheFly to do on-the-fly, xor MatrixVectorStored, to do the stored computation.

Diagonalization of H (continued)

```
// member of MatrixVectorOnTheFly.h
  void matrixVectorProduct(SomeVectorType &x,
                           SomeVectorType const &y) const {
  model_->matrixVectorProduct(x,y,*modelHelper_); }

// in ModelBase.h, matrixVectorProduct
return modelCommon_->matrixVectorProduct(x,y,modelHelper);

// in ModelCommon.h
void matrixVectorProduct(VectorType& x,
                         VectorType& y,
            const ModelHelperType& modelHelper) const {
  //! contribution to Hamiltonian from current system
  modelHelper.hamiltonianLeftProduct(x,y);
  //! contribution to Hamiltonian from current envirnoment
  modelHelper.hamiltonianRightProduct(x,y);
  //! contribution to Hamiltonian from connection S-E
  hamiltonianConnectionProduct(x,y,modelHelper);
}
```

Diagonalization of H (continued)

Let H_m be the Hamiltonian connection between basis2 and basis3 in the orderof basis1 for block m. Then the function hamiltonianConnectionProduct in ModelCommon does $x+ = H_m * y$, where x, and y are vectors.

Algorithm for the connection is:

- Loop over all sites and ask the model class for which connections are active between system and environment. ModelCommon:222 calls HamiltonianConnection.h compute function, which in turn calls calcBond and linkProduct in HamiltonianConnection.h.
- Connections include which sites i in system and j in environment are connected. It also includes which operators A and B are involved. A and B are chosen in getKron in HamiltonianConnection.h.
- calcBond calls modelHelper_.fastOpProdInter for matrixBlock (that is, the heavy version)
- linkProduct calls modelHelper_.fastOpProdInter for vectors (that is, the light version)

Eigenvector of block diagonal matrix

Truncation (In Truncation.h)

```
void changeBasis(... sBasis,... eBasis,...) { // INFINITE
  changeBasis(sBasis,target,keptStates,EXPAND_SYSTEM);
  changeBasis(eBasis,target,keptStates,EXPAND_ENVIRON);

  truncateBasisSystem(sBasis,lrs_.right());
  truncateBasisEnviron(eBasis,lrs_.left());
}

void operator()(...) { // FINITE (discussed later)
  if (direction==EXPAND_SYSTEM) {
     changeBasis(pS,target,keptStates,direction);
     truncateBasisSystem(pS,lrs_.right());
  } else {
     changeBasis(pE,target,keptStates,direction);
     truncateBasisEnviron(pE,lrs_.left());
  }
}
```

Truncation: The Density Matrix

Let us define the density matrix for system. These are square matrices of rank n_l.

$$(\rho^S)_{\alpha,\alpha'} = \sum_{\beta \in \mathcal{V}(E')} \psi^*_{\alpha',\beta} \psi_{\alpha,\beta}$$

ρ^S is matrix in the "system" or left block. ψ is the lowest eigenvector of H computed in the previous step. Some packing is assumed in $\psi_{\alpha,\beta} = \psi_{P(\alpha,\beta)}$. Likewise, the density matrix in the environment is square of rank n_r.

$$(\rho^E)_{\beta,\beta'} = \sum_{\alpha \in \mathcal{V}(S')} \psi^*_{\alpha,\beta'} \psi_{\alpha,\beta}$$

ρ^E is matrix in the "environment" or right block.
This is done in Truncation.h line 207 in function changeBasis, which calls the constructor of DensityMatrixLocal.h.

Diagonalization of The Density Matrix

We then diagonalize ρ^S, and obtain *all* its eigenvalues and eigenvectors, $W^S_{\alpha,\alpha'}$ in $\mathcal{V}(S')$ ordered in decreasing eigenvalue order.
W^S is truncated by using two *alternative* methods:

- We truncate W to a maximum size, say m, xor
- We truncate W using the eigenvalues of ρ up to an error ϵ.

The truncated W, let's call it $\bar{W}$, is square of rank m.
We change basis for the operator $H^{S'}$ (and the c^γ_L operators),

$$(H^{S'\text{new basis}})_{\alpha,\alpha'} = (W^S)^{-1}_{\alpha,\gamma}(H^{S'})_{\gamma,\gamma'}W^S_{\gamma',\alpha'}. \tag{6}$$

Repeat for the environment in a similar way.
This is done in Truncation.h line 229 ff. in function changeBasis, which calls changeBasis in BasisWithOperator.h and Basis.h.

Actual Truncation (in Truncation.h)

```
void truncateBasisSystem(...rSprime,...eBasis) {
  TruncationCache& cache = leftCache_;

  rSprime.truncateBasis(ftransform_,cache.transform,cache.eigs,
                        cache.removedIndices,...);
  // wft work [omitted]
}
```

In BasisWithOperators and Basis the member function truncateBasis will remove the indices passed for all its members (quantumNumbers, operators, etc).

Truncation for the Finite Loops

```
void changeTruncateAndSerialize(...) { // In DmrgSolver.h
  truncate_(...); // operator() in Truncate.h
  if (direction==EXPAND_SYSTEM) {
     checkpoint_.push(lrs_.left());
  } else {
     checkpoint_.push(lrs_.right());
  }

  // serialize [omitted]
}
```

Go to 1st truncation slide to discuss truncation for the finite loops.

White, S. R.,
Phys. Rev. Lett. **69** (1992) 2863.

White, S. R.,
Phys. Rev. B **48** (1993) 345.

Chiara, G. D., Rizzi, M., Rossini, D., and Montangero, S.,
J. Comput. Theor. Nanosci. **5** (2008) 1277.

Schollwöck, U.,
Rev. Mod. Phys. **77** (2005) 259.

Hallberg, K.,
Adv. Phys. **55** (2006) 477.

Rodriguez-Laguna, J.,
http://arxiv.org/abs/cond-mat/0207340, Real Space Renormalization Group Techniques and Applications, 2002.

Alvarez, G.,

Computer Physics Communications **180** (2009) 1572.