DMRG++ v3 Manual

Manual Version: February 9, 2015

Oak Ridge, 2014

Gonzalo Alvarez
Nanomaterials Theory Institute
Oak Ridge National Laboratory

Oak Ridge, TN 37831
February 9, 2015

Contents

http://commons.wikimedia.org/wiki/File:Under_construction_icon-blue.svg

Chapter 1

Quick Start

1.1 Licensing

The full software license for DMRG++ version 2.0.0 can be found in file LICENSE in the root directory of the code. DMRG++ is a free and open source implementation of the DMRG algorithm. You are welcomed to use it and publish data obtained with DMRG++. If you do, **please cite this work** (see next subsection).

1.2 DISCLAIMER

```
THE SOFTWARE IS SUPPLIED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER, CONTRIBUTORS, UNITED STATES GOVERNMENT,
OR THE UNITED STATES DEPARTMENT OF ENERGY BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

NEITHER THE UNITED STATES GOVERNMENT, NOR THE UNITED
STATES DEPARTMENT OF ENERGY, NOR THE COPYRIGHT OWNER, NOR
ANY OF THEIR EMPLOYEES, REPRESENTS THAT THE USE OF ANY
```

```
 INFORMATION, DATA, APPARATUS, PRODUCT, OR PROCESS
 DISCLOSED WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.
```

1.3 How To Cite This Work

```
@article{re:alvarez0209,
author="G. Alvarez",
title="The Density Matrix Renormalization Group for
Strongly Correlated Electron Systems: A Generic Implementation",
journal="Computer Physics Communications",
volume="180",
pages="1572-1578",
year="2009"}

@article{re:alvarez0310,
author="G. Alvarez",
title="Implementation of the SU(2) Hamiltonian
Symmetry for the DMRG Algorithm",
journal="Computer Physics Communications",
volume="183",
pages="2226-2232",
year="2012"}


@article{re:alvarez0311,
author="G. Alvarez and  L. G. G. V. Dias da Silva and
E. Ponce and  E. Dagotto",
title="Time Evolution with the DMRG Algorithm:
A Generic Implementation
for Strongly Correlated Electronic Systems",
journal="Phys. Rev. E",
volume="84",
pages="056706",
year="2011"}

@article{re:alvarez0713,
author="G. Alvarez",
title="Production of minimally entangled typical thermal states
with the Krylov-space approach",
journal="Phys. Rev. B",
volume="87",
pages="245130",
year="2013"}

And also:
```

```
@article{re:alvarez08,
 re:webDmrgPlusPlus,
 Author = {G. Alvarez},
 Title = {DMRG++ Website},
 Publisher = {\\url{http://www.ornl.gov/~gz1/dmrgPlusPlus}} }
```

1.4 Code Integrity

Hash of the latest commit is also posted at
https://web.ornl.gov/ gz1/hashes.html

1.5 Building and Running DMRG++

1.5.1 Required Software

- Item GNU C++
  - Item (required) The LAPACK library.

The configure.pl script will ask for the LDFLAGS variable to pass to the compiler/linker. If the linux platform was chosen the default/suggested LDFLAGS will include -llapack. If the osx platform was chosen the default/suggested LDFLAGS will include -framework Accelerate. For other platforms the appropriate linker flags must be given. More information on LAPACK is at http://netlib.org/lapack/

- Item (required) PsimagLite.

This is here https://github.com/g1257/PsimagLite/. You can do `git clone https://github.com/g1257/PsimagLite.gi` in a separate directory outside of the DMRG++ distribution. 'configure.pl' will ask you where you put it.

- Item (optional) make or gmake (only needed to use the Makefile)
- Item (optional) perl (only needed to run the configure.pl script)

1.5.2 Building DMRG++

```
cd PsimagLite/lib
perl configure.pl
(you may now edit Config.make)
make
cd ../../
cd dmrgpp/src
perl configure.pl
(you may now edit Config.make)
make
```

1.5.3 Running DMRG++

```
./dmrg -f input.inp
```

Sample input files can be found under `TestSuite/inputs/`.

`configure.pl` creates the `Makefile` according to the answers to questions given. In the Makefile, LDFLAGS must contain the linker flags to link with the LAPACK library. Defaults provided automatically by configure.pl should work in most cases. If MPI is not selected (serial code) then the compiler will be chosen to be g++. Other compilers may work but only the GNU C++ compiler, g++, was tested. If MPI is selected then the compiler will be chosen to be mpicxx, which is usually a wrapper script around g++ to take care of linking with MPI libraries and to include MPI headers. Depending on your MPI installation you might need to change the name of this script.

Chapter 2

Input

There is a single input file that is passed as the argument to `-f`, like so

```
./dmrg -f input.inp.
```

Examples of input files can be found under `TestSuite/inputs/` There are three kinds of parameters in the input file: (i) model connections ("geometry") parameters, (ii) model on-site parameters, and (iii) DMRG Solver parameters. Each type of input parameters is discussed below.

2.1 Geometry Input

This needs to be in PsimagLite.

2.2 Model Input

The Model parameters vary from model to model.

2.3 DMRG Solver parameters

Model=string A string indicating the model, be it HubbardOneBand HeisenbergSpinOneHalf, etc.

Options=string A comma-separated list of strings. At least one of the following strings must be provided:

- none Use this when no options are given, since the list of strings must be non-null. Note that "none" does not disable other options.
- useSu2Symmetry Use the SU(2) symmetry for the model, and interpret quantum numbers in the line "QNS" appropriately.
- nofiniteloops Don't do finite loops, even if provided under "FiniteLoops" below.

version=string A mandatory string that is read and ignored. Usually contains the result of doing `git rev-parse HEAD`.

outputfile=string The output file. This file will be created if non-existent, and if it exits it will be truncated.

Value	Description
0	Don't save, compute the ground state
1	Save, compute the ground state
2	Don't save, WFT the ground state
3	Save, WFT the ground state

finiteLoopKeptStates=integer m value for the infinite algorithm.

FiniteLoops=vector A series of space-separated numbers. More than one space is allowed. The first number is the number of finite algorithm movements, followed by series of three numbers for each movement. Of the three numbers, the first is the number of sites to go forward if positive or backward if negative. The second number is the m for this movement and the last number is either 0 or 1, 0 will not save state data to disk and 1 will save all data to be able to calculate observables. The first movement starts from where the infinite loop left off, at the middle of the lattice. See the below for more information and examples on Finite Loops.

TargetElectronsUp=integer

TargetElectronsDown=integer

2.3.1 Finite Loops

2.3.2 Enabling finite loops

To enable finite loops make sure that the option 'nofiniteloops' is *not* present under 'SolverOptions='. Remember that the entry `FiniteLoops` in the input file is a series of space-separated numbers. More than one space is allowed. The first number is the number of finite algorithm "movements," followed by series of three numbers for each movement. Of the three numbers, the first is the number of sites to go forward if positive or backward if negative. The second number is the m for this movement and the last number is either 0 or 1, 0 will not save state data to disk and 1 will save all data to be able to calculate observables. The first movement starts from where the infinite loop left off, at the middle of the lattice.

2.3.3 Example of a Finite loops line in the input file

```
FiniteLoops 4 7 200 0 -7 200 0 7 200 1 7 200 1
```

The number 4 implies 4 finite loops. The first fine loop is `7 200 0`, meaning go forward 7 steps, use m=*200* for this finite sweep, and 0: do not store transformation in disk. The next is `-7 200 0`, which goes backwards 7 sites, etc. Remember that the finite loops start at the middle of the lattice, where the infinite loop left off. ADD FIGURE SHOWING WHAT THIS DOES.

2.3.4 The third number in the triplet

The save option is a bitwise option where the first bit means save or don't save, and the second bit compute the g.s. or WFT it. So there are 4 combinations (as of today):

2.3.5 Caveats and Troubleshooting

If 'nofiniteloops' is an option in the options line of the input file then the `FiniteLoops` line in the input file is ignored, and no finite loops are done. In this case, DMRG++ stops when the infinite algorithm has finished.

Make sure the first number is the number of triplets that follow.

Make sure you don't fall off the lattice, by going forward or backwards too much. Remember that at least one site must remain for the system part of the lattice. So on a 16 site chain, when you start the finite loops you're at the middle, you can go forward at most 7 sites, and backwards at most 7 sites.

There is some checking done to the finite loops input, see PTEXREF139, but you might find that it's not comprehensive.

Chapter 3

Output

3.1 Standard Output and Error

If you run

```
./dmrg -f input.inp
```

you will see messages printed to the terminal. These are process by the `PsimagLite` class `ProgressIndicator` and are designed to show the DMRG++ progress. All these messages are of the form:

```
Class [T]: Message
```

where `Class` is the class that is currently executing and the message hints at what is being executed. The number `T` between brackets is the wall time ellapsed since program start.

3.2 The Data file

3.3 The Disk Stacks

3.4 The Wft Data Files

3.5 Signals

PLEASE NOTE: This is an experimental (CITATION NEEDED FIXME) feature. To use it you must add `-DUSE_SIGNALS` to `CPPFLAGS` in the Makefile.

3.5.1 SIGUSR1

Rationale: When running a process in a queue batching system the standard output and standard error might be buffered, and, thus, might not be seen until program completion. DMRG++ allows the user to store (a fragment of) the stdout and stderr buffers into a temporary file to monitor program process in situations where stdout and stderr would not normally be accessible.

Sending the signal SIGUSR1 to the DMRG++ process will result in switching the state of the ProgressIndicator buffer: if the state was inactive it will become active, and viceversa. Only when the state of the ProgressIndicator buffer is active does ProgressIndicator store its stream in memory. This stream contains the standard output and standard error printed by DMRG++. When the state of the ProgressIndicator is switched back from active to inactive, DMRG++ dumps the buffer into a temporary file, and closes the buffer. The temporary file is named bufferN.txt where N is the PID of the DMRG++ process.

HINT: qsig might be used to send a signal if the DMRG++ process is running in PBS or torque.

CAVEATS: Leaving the buffer on for long periods of time might cause high memory consumption. The temporary buffer file is overwritten if the buffer is used more than once by the same process. The temporary buffer file is not deleted at the end of program execution.

Chapter 4

Developer's Guide

4.1 Main Driver

The high level program is this

```
//! Setup the Model
ModelType model(mp,dmrgGeometry);

//! Setup the dmrg solver:
SolverType dmrgSolver(dmrgSolverParams,model,concurrency);

//! Perform DMRG Loops:
dmrgSolver.main();
```

Graphic showing the template dependencies of the classes

4.2 DMRG Engine

4.2.1 DMRG Algorithm

Let us define *block* to mean a finite set 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 Heisenberg model by $C = \{\uparrow, \downarrow\}$. A *real-space-based Hilbert space* $\mathcal{V}$ on a block B and set C is a Hilbert space with basis B^C. I will simply denote this as $\mathcal{V}(B)$ and assume that C is implicit and fixed. A *real-space-based Hilbert space* can also be thought of as the external product space of $\#B$ Hilbert spaces on a site, one for each site in block B. We will consider general Hamiltonians that act on Hilbert spaces $\mathcal{V}$, as previously defined.

I shall give a procedural description of the DMRG method 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. This is shown schematically in Fig. 4.1. All sites in S, X, Y and E are numbered as shown in the figure.

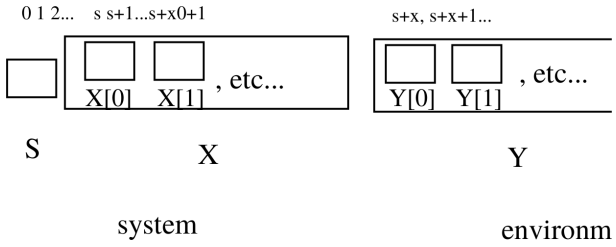

Figure 4.1: Labeling of blocks for the DMRG procedure. Blocks from vector of blocks X are added one at a time to block S to form the system and blocks from vector of blocks Y are added one at a time to E to form the environment. Blocks are vectors of integers. The integers (numbers at the top of the figure) label all sites in a fixed and unique way.

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.

Consider $\hat{H}_{S' \cup E'}$, the Hamiltonian operator, acting on $\mathcal{V}(S') \otimes \mathcal{V}(E')$. Using Lanczos4.4.3, we diagonalize $\hat{H}_{S' \cup E'}$ to obtain its lowest eigenvector:

$$|\psi\rangle = \sum_{\alpha \in \mathcal{V}(S'), \beta \in \mathcal{V}(E')} \psi_{\alpha,\beta} |\alpha\rangle \otimes |\beta\rangle, \tag{4.1}$$

where $\{|\alpha\rangle\}$ is a basis of $\mathcal{V}(S')$ and $\{|\beta\rangle\}$ is a basis of $\mathcal{V}(E')$.

We proceed in the same way for the environment, diagonalize $\hat{\rho}_E$ to obtain ordered eigenvectors w^E, and define $(H^{E' \text{new basis}})_{\alpha,\alpha'}$. Now we set $S \leftarrow S'$, $\mathcal{V}_R(S) \leftarrow \mathcal{V}_R(S')$, $H_{S'} \leftarrow H_S$, and similarly for the environment, increase step by one, 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 (Section to be written) 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.

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[1] 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 [1] $\epsilon_{tr} < 10^{-6}$. 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 [1, 6, 2, 5].

4.2.2 Driver Program

Let us motivate the discussion by introducing a typical problem to be solved by DMRG: "Using the DMRG method, one would like to calculate the local density of states on all sites for a Hubbard

model with inhomogeneous Hubbard U values on a one-dimensional (1D) chain". We want to modularize as many tasks mentioned in the last sentence as possible. We certainly want to separate the DMRG solver from the model in question, since we could later want to do the same calculation for the t-J model; and the model from the lattice, since we might want to do the same calculation on, say, a n-leg ladder, instead of a 1D chain. C++ is a computer language that is very fit for this purpose, since it allows to template classes. Then we can write a C++ class to implement the DMRG method (DmrgSolver class), and template this class on a strongly-correlated-electron (SCE) model template, so that we can delegate all SCE model related code to the SCE model class.

However, for DmrgSolver to be able to use a given SCE model, we need a convention that such SCE model class will have to follow. This is known as a C++ public interface, and for a SCE model it is given in DmrgModelBase. To do the calculation for a new SCE model, we simply need to implement all functions found in DmrgModelBase *without* changing the DmrgSolver class. The model will, in turn, be templated on the geometry. For example, the Hubbard model with inhomogeneous Hubbard U values and inhomogeneous hoppings (class DmrgModelHubbard) delegates all geometry related operations to a templated geometry class. Then DmrgModelHubbard can be used for, say, one-dimensional chains and n-leg ladders *without* modification. This is done by just instantiating DmrgModelHubbard with the appropriate geometry class, either DmrgGeometryOneD or DmrgGeometryLadder, or some other class that the reader may wish to write, which implements the interface given in DmrgGeometryBase. Add figure showing interfaces

In the following sections I will describe these different modules. Since the reader may wish to first understand how the DMRG method is implemented, I will start with the core C++ classes that implement the method. The user of the program will not need to change these core classes to add functionality. Instead, new models and geometries can be added by creating implementations for DmrgModelBase and DmrgGeometryBase, and those public interfaces will be explained next.

But for now I end this section by briefly describing the "driver" program for a Hubbard model on a 1D chain (see file `dmrg.cpp`). There, DmrgSolver is instantiated with DmrgModelHubbard, since in this case one wishes to perform the calculation for the Hubbard model. In turn, DmrgModelHubbard is instantiated with DmrgGeometryOneD since now one wishes to perform the calculation on a 1D chain.

Expand the driver explanation

4.2.3 DmrgSolver and The "Infinite" DMRG Algorithm

The purpose of the DmrgSolver class is to perform the loop for the DMRG "infinite algorithm" discussed before. This class also performs the "finite algorithm" [6] to allow for the calculation of static (and in the future dynamic) observables, such as static correlations.

The program is structured as a series of header files containing the implementation[1] with each class written in the header file of the same name, and a "driver" program that uses the capabilities provided by the header files to solve a specific problem. To simplify the discussion, we start where the "driver program" starts, in its `int main()` function, which calls `dmrgSolver.main()`, whose main work is to perform the loop for the "infinite" DMRG algorithm. Let us now discuss this loop which is found in the `infiniteDmrgLoop` function, and is sketched in Fig. 4.2.

[1] Traditionally, implementation is written in cpp files that are compiled separately. However, here templates are used heavily, and to avoid complications related to templates that some C++ compilers cannot handle, we choose to have only one translation unit.

```
for (step=0;step<X.size();step++) {
  // grow system (a)
  lrs_.growLeftBlock(model_,pS,X[step]);
  // grow environment (b)
  lrs_.growRightBlock(model_,pE,Y[step]);
  // product of system and environment (c)
  lrs_.setToProduct(quantumSector_);

  diagonalization_(psi,INFINITE,X[step],Y[step]); (d)
  truncate_(pS,psi,parameters_.keptStatesInfinite,EXPAND_SYSTEM); (e)
  truncate_(pE,psi,parameters_.keptStatesInfinite,EXPAND_ENVIRON); (f)

  checkpoint_.push(pS,pE); //(g)
}
```

Figure 4.2: Implementation of the "infinite" DMRG loop for a general SCE model on a general geometry.

In Fig. 4.2(a) the system pS is grown by adding the sites contained in block X[step]. Note that X is a vector of blocks to be added one at a time[2]. The block X[step] (usually just a single site) is added *to the right of* pS. The result is stored in `lrs_.left()`. Similarly is done in Fig. 4.2(b) for the environment: the block Y[step] (usually just a single site) is added to the environment given in pE and stored in `lrs_.right()`. This time the addition is done *to the left of* pE, since pE is the environment. In Fig. 4.2(c) the outer product of `lrs_.left()` (the new system) and `lrs_.right()` (the new environment) is made and stored in pSE. The actual task is delegated to the Basis class (see Section 4.3). In Fig. 4.2(d) the diagonalization of the Hamiltonian for block pSE is performed, and the ground state vector is computed and stored in psi, following Eq. (4.1). The object called concurrency is used to handle parallelization over matrix blocks related to symmetries present in the model (see section). Next, in Fig. 4.2(e) the bases are changed following Eqs. (4.5,4.6,4.7), truncated if necessary, and the result is stored in pS for the system, and in pE, Fig. 4.2(f), for the environment. Note that this overwrites the old pS and pE, preparing these variable for the next DMRG step.

A copy of the current state of the system is pushed into a last-in-first-out stack in Fig. 4.2(g), so that it can later be used in the finite DMRG algorithm (not discussed here, see code). The loop continues until all blocks in vector of blocks X have been added to the initial system S, and all blocks in vector of blocks Y have been added to the initial environment E. We repeat again that vector of sites are used instead of simply sites to generalize the growth process, in case one might want to add more than one site at a time.

The implementation of the steps mentioned in the previous paragraph (i.e., growth process, outer products, diagonalization, change of basis and truncation) are described in FIXME.

[2] So X is a vector of vector of integers, and X[step] is a vector of integers.

4.2.4 Finite Algorithm

4.3 Hilbert Space Basis I: DmrgBasis and Symmetries

4.3.1 Local Symmetries

DMRG++ has two C++ classes that handle the concept of a basis (of a Hilbert space). The first one (DmrgBasis) handles reordering and symmetries in a general way, without the need to consider operators. The second one (DmrgBasisWithOperators) does consider operators, and will be explained in the next sub-section. The advantage of dividing functionality in this way will become apparent later.

In any actual computer simulation the "infinite" DMRG loop will actually stop at a certain point. Let us say that it stops after 50 sites have been added to the system[3]. There will also be at this point another 50 sites that constitute the environment. Now, from the beginning each of these 100 sites is given a fixed number from 0 to 99. Therefore, sites are always labeled in a fixed way and their labels are always known (see Fig. 4.1). The variable block of a DmrgBasis object indicates over which sites the basis represented by this object is being built.

Symmetries will allow the solver to block the Hamiltonian matrix in blocks, using less memory, speeding up the computation and allowing the code to parallelize matrix blocks related by symmetry. Let us assume that our particular model has N_s symmetries labeled by $0 \leq \alpha < N_s$. Therefore, each element k of the basis has N_s associated "good" quantum numbers $\tilde{q}_{k,\alpha}$. These quantum numbers can refer to practically anything, for example, to number of particles with a given spin or orbital or to the z component of the spin. We do not need to know the details to block the matrix. We know, however, that these numbers are finite, and let Q be an integer such that $\tilde{q}_{k,\alpha} < Q \ \forall k, \alpha$. We can then combine all these quantum numbers into a single one, like this: $q_k = \sum_\alpha \tilde{q}_{k,\alpha} Q^\alpha$, and this mapping is bijective. In essence, we combined all "good" quantum numbers into a single one and from now on we will consider that we have only one Hamiltonian symmetry called the "effective" symmetry, and only one corresponding number q_k, the "effective" quantum number. These numbers are stored in the member *quantumNumbers* of C++ class Basis. (Note that if one has 100 sites or less,[4] then the number Q defined above is probably of the order of hundreds for usual symmetries, making this implementation very practical for systems of correlated electrons.)

We then reorder our basis such that its elements are given in increasing q number. There will be a permutation vector associated with this reordering, that will be stored in the member `permutationVector` of class Basis. For ease of coding we also store its inverse in `permInverse`.

What remains to be done is to find a partition of the basis which 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 4x4, then of a block of 2x2, etc.

[3] For simplicity, this explanatory text considers the case of blocks having a single site, so one site is added at a time, but a more general case can be handled by the code.

[4] This is probably a maximum for systems of correlated electrons such as the Hubbard model or the t-J model.

The quantum numbers of the original (untransformed) real-space basis are set by the model class (to be described in Section 4.5), whereas the quantum numbers of outer products are handled by the class Basis and BasisImplementation, 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.

4.3.2 Product of Spaces

If $\mathcal{V}_1$ is a Hilbert space of dimension n_1, and $\mathcal{V}_2$ is a Hilbert space of dimension n_2, then a state $\psi \in \mathcal{V}_1 \otimes \mathcal{V}_2$ is given by: $\psi_{\alpha,\beta}$ with $\alpha \in \mathcal{V}_1$ and $\beta \in \mathcal{V}_2$. In DMRG++ a single index is used to "pack" α and β together, like this $\alpha + n_1\beta$ (you can prove that this is a bijection from $(\alpha, \beta) \longleftrightarrow \alpha + n_1\beta$. This isn't the complete packing, however, because we need to reorder the states for symmetry reasons. We use then a permutation (actually it's the inverse permutation in the code) P^{-1}. Putting all together you get ψ_i, with $i = P^{-1}(\alpha + n_1\beta)$. I like to write it more formally with a Kronecker delta, like this: $\sum_i \psi_i \delta_{P(i),\alpha+n_1\beta}$. Note how, for fixed α and β, the δ picks up the correct (packed and permuted) index i.

4.3.3 Left, Right, and Super

In standard DMRG, states are decomposed into left, right and super[block] spaces. The left and right spaces are further decomposed into a block plus a site, on the left side; and site plus a block, on the right side. There are then 3 products of spaces, giving rise to 3 packings and 3 permutations. I use the notations P_S, P_E, and P_{SE}, respectively (or its inverses). The words left and system, right and environment, and super[block] and system-environment are used interchangably in the code.

Consider the system or left block $\mathcal{S}$ as $\mathcal{S}'$ plus a site, like this: $\mathcal{S} = \mathcal{S}' \otimes \mathcal{V}_1$. Consider an operator $A_{x_1,x_1'}$ acting on the space of one site, $\mathcal{V}_1$ such that both $x_1, x_1' \in \mathcal{V}_1$. Consider the superblock $\mathcal{S}' \otimes \mathcal{V}_1 \otimes \mathcal{E} \equiv \mathcal{S} \otimes \mathcal{E}$. Consider a state on the superblock ψ. Then the $j-$th component of $A\psi$ is

$$(A\psi)_j = \sum \delta_{P_{SE}(i),x+yn_s}\delta_{P_S(x),x0+x1n_0)}A_{x_1,x_1'} \tag{4.2}$$

$$\delta_{x',P_S^{-1}(x0+x1'n_0)}\delta_{j,P_{SE}^{-1}(x'+y'n_s)}\psi_i f_S(x) \tag{4.3}$$

A sum over all indices except j is assumed. Let's analyze this. Because i is in the superblock, it's packed and permuted, so we have to unpack it and un-permute it, that is, find $x \in \mathcal{S}$ and $y \in \mathcal{E}$ that correspond to it. Formally, in the equation in this paper, the delta picks up the right one: $\delta_{P_{SE}(i),x+yn_s}$. In the code we simply apply the permutation P_{SE} to i and then divide by n_s. The quotient is y and the remainder is x. Now, A does not act on $\mathcal{E}$, so y will remain the same. But $x \in \mathcal{S}' \otimes \mathcal{V}_1$, so again, we need to unpack it and un- permute it. That's what the next delta, $\delta_{P_S(x),x0+x1n_0}$ does. Note that now the permutation is P_S as opposed to P_{SE}. We are ready now to apply A, with $A_{x_1,x_1'}$.

Next, the reverse procedure must be applied. We need to pack x_0, x_1', y into a single index and perform all permutations. We do this in two stages (two δs). The first one does x_0, x_1' into x', the second one x', y into j, which is the free index.

In the above, I left out a few complications, which I will now attend to.

Fermionic Sign

To apply $A_{x_1,x_1'}$ to (x_0, x_1) we need to step over x_0, that is, the space $\mathcal{S}'$. If A is fermionic we might pick up a sign. This sign will be equal to 1 if the number of electrons of state x_0 is even, and -1 otherwise. This is represented by $f_S(x)$ in the equation above.

Vectors in chunks

Because of symmetry, the vector ψ has only one (or a few) non-zero chunks. Then we ought not to loop or sum over the whole superblock like in $\sum_i$. Instead, we need to only loop or sweep or sum over the non-zero chunk(s). This is easily taken care of by storing the partition of the superblock, and is mostly straightforward, except that there is one complication we must attend to. For simplicity, assume that ψ has only one chunk. Then if the resulting j is outside ψ's chunk, $A\psi$ and ψ will have different symmetries, and $A\psi$ will need to be stored in a different chunk. The DMRG++ class `VectorWithOffsets` does this transparently. For performance reasons, there is also a `VectorWithOffset` class to use when we know A does not transport ψ into a different symmetry chunk.

Sparse matrices

The matrix A is usually stored in a sparse format. DMRG++ uses compressed row storage (CRS, REFERENCE HERE FIXME). Therefore the looping is done according to the CRS scheme. See the loop in `ApplyOperatorLocal.h`, line 155 or whereabouts.

4.3.4 SU(2) Symmetry

4.4 Hilbert Space Basis II: DmrgBasisWithOperators

4.4.1 Outer Product of Operators

A class to represent a Hilbert Space for a strongly correlated electron model Derives from Basis

C++ class Basis (and BasisImplementation) implement only certain functionality associated with a Hilbert space basis, as mentioned in the previous section. However, more capabilities related to a Hilbert space basis are needed.

C++ class BasisWithOperators inherits from Basis, and contains certain local operators for the basis in question, as well as the Hamiltonian matrix. The operators that need to be considered here are operators necessary to compute the Hamiltonian across the system and environment, and to compute observables. Therefore, the specific operators vary from model to model. For example, for the Hubbard model, we consider $c_{i\sigma}$ operators, that destroy an electron with spin σ on site i. For the Heisenberg model, we consider operators S_i^+ and S_i^z for each site i. In each case these operators are calculated by the model class (see Section 4.5) on the "natural" basis, and added to the basis in question with a call to `setOperators()`. These local operators are stored as sparse matrices to save memory, although the matrix type is templated and could be anything. For details on the implementation of these operators, see OperatorsBase, its common implementation OperatorsImplementation, and the two examples provided OperatorsHubbard and OperatorsHeisenberg for the Hubbard and Heisenberg models, respectively. Additionally, BasisWithOperators has a number of member functions to handle operations that the DMRG method performs on local operators in a

Hilbert space basis. These include functions to create an outer product of two given Hilbert spaces, to transform a basis, to truncate a basis, etc.

Let us now go back to the "infinite" DMRG loop and discuss in more detail Fig. 4.2(a) ((b) is similar)), i.e., the function grow().

Local operators are set for the basis in question with a call to BasisWithOperators's member function `setOperators()`. 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 (pSprime,pS,Xbasis,dir,option)` in the grow function, which actually calls `pSprime.setToProduct (pS,xBasis,dir)` 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 dmrgBasis object for the outer product, pSprime, is set to contain this full Hamiltonian with the call to `pSprime.setHamiltonian(matrix)`.

I will know explain how the full outer product between two operators is implemented. If local operator A lives in Hilbert space $\mathcal{A}$ and local operator B lives in Hilbert space $\mathcal{B}$, then $C = AB$ lives in Hilbert space $\mathcal{C} = \mathcal{A} \otimes \mathcal{B}$. Let α_1 and α_2 represent states of $\mathcal{A}$, and let β_1 and β_2 represent states of $\mathcal{B}$. Then, in the product basis, $C_{\alpha_1,\beta_1;\alpha_2,\beta_2} = A_{\alpha_1,\alpha_2} B_{\beta_1,\beta_2}$. Additionally, $\mathcal{C}$ is reordered such that each state of this outer product basis is labeled in increasing effective quantum number (see Section 4.3). In the previous example, if the Hilbert spaces $\mathcal{A}$ and $\mathcal{B}$ had sizes a and b, respectively, then their outer product would have size ab. When we add sites to the system (or the environment) the memory usage remains bounded by the truncation, and it is usually not a problem to store full product matrices, as long as we do it in a sparse way (DMRG++ uses compressed row storage). In short, local operators are always stored in the most recently transformed basis for *all sites* and, if applicable, *all values* of the internal degree of freedom σ. See PTEXREFsetToProductOps and PTEXREFHERE.

The Operators class stores the local operators for this basis. Only the local operators corresponding to the most recently added sites will be meaningful. Indeed, if we apply transformation W (possibly truncating the basis, see Eq. (4.7)) then

$$(W^\dagger A W)(W^\dagger B W) \neq W^\dagger (AB) W, \tag{4.4}$$

since $WW^\dagger \neq 1$ because the DMRG truncation does not assure us that $W^\dagger$ will be the right inverse of W (but $W^\dagger W = 1$ always holds). Because of this reason we cannot construct the Hamiltonian simply from transformed local operators, even if we store them for all sites, but we need to store also the Hamiltonian in the most recently transformed basis. The fact that Operators stores local operators in the most recently transformed basis for *all sites* does not increase memory usage too much, and simplifies the writing of code for complicated geometries or connections, because all local opeators are availabel at all times. Each SCE model class is responsible for determining whether a transformed operator can be used (or not because of the reason limitation above).

Let us now examine in more detail Fig. 4.2(c), where we form the outer product of the current system and current environment, and calculate its Hamiltonian. We could use the same procedure as outlined in the previous paragraph, i.e., to use the DmrgBasisWithOperators class to resize the matrices for all local operators. Storing matrices in this case (even in a sparse way and even considering that there is truncation) would be too much of a penalty for performance. Therefore, in this latter case we do the outer product on-the-fly only, without storing any matrices. In Fig. 4.2(c) pSE contains the outer product of system and environment, but pSE is only a Basis object, not a

BasisWithOperators object, i.e., it does not contain operators. In the code see `setToProductSolver`, and `setToProductLrs`.

We now consider Fig. 4.2(d), where the diagonalization of the system's plus environment's Hamiltonian is performed. Since `lrs_.super()`, being only a Basis object, does not contain all the information related to the outer product of system and environment (as we saw, this would be prohibitively expensive), we need to pass the system's basis (pSprime) and the environment's basis (pEprime) to the diagonalization functor `diagonalization_()`, see `Diagonalization`, in order to be able to form the outer product on-the-fly. There, since `lrs_.super()` does provide information about effective symmetry blocking, we block the Hamiltonian matrix using effective symmetry, and call `diagonaliseOneBlock()`, see `diagonaliseOneBlock`, for each symmetry block. Only those matrix blocks that contain the desired or targeted number of electrons (or other local symmetry) will be considered.

4.4.2 Truncation

Let us define the density matrices for system:

$$(\hat{\rho}_S)_{\alpha,\alpha'} = \sum_{\beta \in \mathcal{V}(E')} \psi^*_{\alpha',\beta} \psi_{\alpha,\beta} \tag{4.5}$$

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

$$(\hat{\rho}_E)_{\beta,\beta'} = \sum_{\alpha \in \mathcal{V}(S')} \psi^*_{\alpha,\beta'} \psi_{\alpha,\beta} \tag{4.6}$$

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

$$(H^{S' \text{new basis}})_{\alpha,\alpha'} = (w^S)^{-1}_{\alpha,\gamma} (H^{S'})_{\gamma,\gamma'} w^S_{\gamma',\alpha'}. \tag{4.7}$$

4.4.3 Lanczos Solver

To diagonalize Hamiltonian H we use the Lanczos method[3, 4], although this is also templated.

For the Lanczos diagonalization method we also want to provide as much code isolation and modularity as possible. The Lanczos method needs only to know how to perform the operation $x+ = Hy$, given vectors x and y. Using this fact, we can separate the matrix type from the Lanczos method. To keep the discussion short this is not addressed here, but can be seen in the `diagonaliseOneBlock()` function, and in classes SolverLanczos, HamiltonianInternalProduct, and DmrgModelHelper. The first of these classes contains an implementation of the Lanczos method that is templated on a class that simply has to provide the operation $x+ = Hy$ and, therefore, it is generic and valid for any SCE model. It is important to remark that the operation $x+ = Hy$ is finally delegated to the model in question. As an example, the operation $x+ = Hy$ for the Hubbard model is performed in function `matrixVectorProduct()` in class DmrgModelHubbard. This function performs only three tasks: (i) $x+ = H_{system}y$, (ii) $x+ = H_{environment}y$ and (iii) $x+ = H_{connection}y$. The fist two are straightforward, so we focus on the last one, in `hamiltonianConnectionProduct()`, that considers the part of the Hamiltonian that connects system and environment. This function

runs the following loop: for every site i in the system and every site j in the environment it calculates $x+ = H_{ij}y$ in function `linkProduct`, after finding the appropriate tight binding hopping value.

The function `linkProduct` is useful not only for the Hubbard model, but it is generic enough to use in other SCE models that include a tight binding connection of the type $c^\dagger_{i\sigma}c_{j\sigma}$, and, therefore, is part of a separate class, ConnectorHopping. Likewise, the function `linkProduct` in ConnectorExchange deals with Hamiltonian connections of the type $\vec{S}_i \cdot \vec{S}_j$, and can be used by models that include that type of term, such as the sample Heisenberg model provided with DMRG++. We remind readers that wish to understand this code that the function `linkProduct` and, in particular, the related function `fastOpProdInter` are more complicated than usual, since (i) the outer product is constructed on the fly, and (ii) the resulting states of this outer product need to be reordered so that effective symmetry blocking can be used.

4.5 Model Interface

4.5.1 Abstract Interface

4.5.2 Heisenberg Model

4.5.3 One-Orbital Hubbard Model

4.5.4 Many-Orbital Hubbard Model

4.5.5 t-J model

4.6 Geometry Interface

4.6.1 Abstract Interface

I present two sample geometries, one for 1D chains and one for n-leg ladders in classes DmrgGeometryOneD and DmrgGeometryLadder. Both derive from the abstract class DmrgGeometryBase. To implement new geometries a new class needs to be derived from this base class, and the functions in the base class (the interface) needs to be implemented. As in the case of DmrgModelBase, the interface is documented in the code, but here I briefly describe the most important functions.

The function `setBlocksOfSites` needs to set the initial block for system and environment, and for the vector of blocks X and Y to be added to system and environment, respectively, according to the convention given in Fig. 4.1. There are two `calcConnectorType` functions. Both calculate the type of connection between two sites i and j, which can be SystemSystem, SystemEnviron, EnvironSystem or EnvironEnviron, where the names are self-explanatory. The function `calcConnectorValue` determines the value of the connector (e.g., tight-binding hopping for the Hubbard model or J_{ij} for the case of the Heisenberg model) between two sites, delegating the work to the model class if necessary. The function `findExtremes` determines the extremes sites of a given block of sites and the function `findReflection` finds the "reflection" in the environment block of a given site in the system block or vice-versa.

4.6.2 One Dimensional Chains

4.6.3 Ladders

4.7 Concurrency Interface: Code Parallelization

4.7.1 Abstract Interface

The Concurrency class encapsulates parallelization. Two concrete classes that implement this interface are included in the present code. One is for serial code (ConcurrencySerial class) that does no parallelization at all, and the other one (ConcurrencyMpi class) is for parallelization based on the MPI[5]. Other parallelization implementations, e.g. using pthreads, can be similarly written by implementing this interface. The interface is described in place in class Concurrency. Here, I briefly mention its most important functions. Function `rank()` returns the rank of the current processor or thread. `nprocs()` returns the total number of processors. Functions `loopCreate()` and `loop()` handle a parallelization of a standard loop. Function `gather()` gathers data from each processor into the root processor.

4.7.2 MPI

4.7.3 Pthreads

4.7.4 CUDA

4.8 Input and Output

4.8.1 Input System

4.8.2 DiskStack

4.8.3 Program Output

4.8.4 Test Suite

4.9 Optimizations

4.9.1 Wave Function Transformation

I will describe the WFT when shrinking the system. The implementation of this is in class WaveFunctionTransformation. Here I focus on the system without SU(2) symmetry support first. We need to consider that site j has just been swallowed by the growing environment. See figure 4.3 for the setup. Latin letters label points, Greek letters label states. The sub-index p indicates the newest DMRG step. The approximate guess for the new wave-function $\psi^p_{\eta_p}$ is given in terms of the previous wave-function ψ_η by:

$$\psi^p_{\eta_p} = W^S_{\alpha,\epsilon}\psi_\eta W^E_{\beta,\beta_p}\delta_{P^{SE}(\eta);\epsilon+\beta n_0}\delta_{P^S(\alpha);\alpha_p+\kappa_p n_1}\delta_{P^S_p(\eta_p);\alpha_p+\gamma_p n_2}\delta_{P^E_p(\gamma_p);\kappa_p+\beta_p n_3}, \tag{4.8}$$

[5] See, for example, http://www-unix.mcs.anl.gov/mpi/

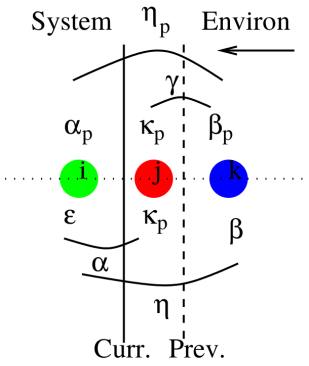

Figure 4.3: TBW.

where the system transformation is W^S, the environment transformation is W^E, the new system-environment permutation P_p^{SE}, the new environment permutation P_p^E, the old system-environment permutation P^{SE}, and the new system permutation P^S. A sum should be assumed for all indices except η_p.

4.9.2 SU(2) Reduced Operators

4.9.3 Checkpointing

Let's say you want to first run 3 moves on a 16-site lattice: `7 100 0 -5 100 0 -2 100 0`, and then continue it with 2 more `-7 200 0 7 200 0`. Then one needs to set up input files as in `TestSuite/inputs/input23.inp` and `TestSuite/inputs/input24.inp`. Compile and do the first run:

```
cd src/
perl configure.pl < ../TestSuite/inputs/model23.spec
(or say ModelHeisenberg and DiskStack)
make
./dmrg ../TestSuite/inputs/input23.inp
```

You are ready for the continuation of this run now with:

```
./dmrg ../TestSuite/inputs/input24.inp
```

Note that the continued run's input (`input24.inp`) has the option checkpoint, and the Checkpoint-Filename tag. Continued (raw) results will be in file `data24` as usual.

Note the following caveat or "todo":

- There's no check (yet) of finite loops for consistency while checkpoint is in use. Therefore, make sure the second run is starting at a point on the lattice where the previous to-be-continued run left off.

4.9.4 Distributed Parallelization

4.9.5 Shared-memory Parallelization

4.10 Static Observables

A quick run with the calculation of static observables can be done like so:

```
cd src/
perl configure.pl
(all default answers here)
make
make observe
./dmrg ../TestSuite/input2.inp
./observe ../TestSuite/input2.inp data2.txt
```

You will see something like this:

```
OperatorC:
8 16
0.5 0.426244 -2.84172e-08 ...
0 0.5 -0.252775 1.67222e-07 0.0586682 ...
...
```

Here we are computing $C_{ij} = \langle c_{i\uparrow} c_{j\uparrow} \rangle$, where 816 are the dimensions of the matrix that follow (C_{ij} is not computed for $i > j$). For example $C_{00} = 0.5$, $C_{01} = 0.426244$, $C_{12} = -0.252775$.

The same is done for $N_{ij} = \langle n_i n_j \rangle$, where $n_i = n_{i\uparrow} + n_{i\downarrow}$, and also for $\langle S^z_i S^z_j \rangle$

The observer driver (`observe.cpp`) controls what is calculated. Please have a look at it and modify as necessary.

THIS SECTION NEEDS MORE WORK. IN PARTICULAR HOW TO SETUP THE INPUT FILE TO BE ABLE TO PRODUCE DATA FOR THE OBSERVER.

4.10.1 Ground State Energy and Error

4.10.2 Static Correlations

4.10.3 Observables Driver

LICENSE

```
Copyright (c) 2009 , UT-Battelle, LLC
All rights reserved

[DMRG++, Version 2.0.0]
[by G.A., Oak Ridge National Laboratory]

UT Battelle Open Source Software License 11242008

OPEN SOURCE LICENSE
```

```
Subject to the conditions of this License, each
contributor to this software hereby grants, free of
charge, to any person obtaining a copy of this software
and associated documentation files (the "Software"), a
perpetual, worldwide, non-exclusive, no-charge,
royalty-free, irrevocable copyright license to use, copy,
modify, merge, publish, distribute, and/or sublicense
copies of the Software.

1. Redistributions of Software must retain the above
copyright and license notices, this list of conditions,
and the following disclaimer.  Changes or modifications
to, or derivative works of, the Software should be noted
with comments and the contributor and organization's
name.

2. Neither the names of UT-Battelle, LLC or the
Department of Energy nor the names of the Software
contributors may be used to endorse or promote products
derived from this software without specific prior written
permission of UT-Battelle.

3. The software and the end-user documentation included
with the redistribution, with or without modification,
must include the following acknowledgment:

"This product includes software produced by UT-Battelle,
LLC under Contract No. DE-AC05-00OR22725  with the
Department of Energy."

************************************************************
DISCLAIMER

THE SOFTWARE IS SUPPLIED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER, CONTRIBUTORS, UNITED STATES GOVERNMENT,
OR THE UNITED STATES DEPARTMENT OF ENERGY BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
```

```
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

NEITHER THE UNITED STATES GOVERNMENT, NOR THE UNITED
STATES DEPARTMENT OF ENERGY, NOR THE COPYRIGHT OWNER, NOR
ANY OF THEIR EMPLOYEES, REPRESENTS THAT THE USE OF ANY
INFORMATION, DATA, APPARATUS, PRODUCT, OR PROCESS
DISCLOSED WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

************************************************************
```

Bibliography

[1] G. De Chiara, M. Rizzi, D. Rossini, and S. Montangero. *J. Comput. Theor. Nanosci.*, 5:1277–1288, 2008.

[2] K. Hallberg. *Adv. Phys.*, 55:477–526, 2006.

[3] C. Lanczos. *J. Res. Nat. Bur. Stand.*, 45:255, 1950.

[4] D.G. Pettifor and D.L. Weaire, editors. *The Recursion Method and Its Applications, Springer Series in Solid-State Sciences*, volume 58. Springer Verlag, Berlin/Heidelberg, 1985.

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

[6] U. Schollwöck. The density-matrix renormalization group. *Rev. Mod. Phys.*, 77:259, 2005.