ITensors in Julia

(Dated: April 2023)

I. INTRODUCTION

ITensor is a software library to develop tensor network algorithms.

A. Installing and Importing

Installation of ITensor requires using Julia v1.3 or later. The installed version of Julia can be seen by

```
julia> VERSION
```

The current stable release is v1.8.5. To install the ITensor package in Julia go to package manager and add the ITensor package.

```
julia> ]
pkg> add ITensors
```

This will install the ITensor package. The package manager can be exited by *ctrl+C* or *backspace*. After installation, the package can be imported in a Julia file by the *using* command

```
using ITensors
```

II. MATRIX PRODUCT OPERATORS

A. Tensor Index

In ITensor, before constructing a tensor object one first needs to create an index object that will represent the indices of the tensors. In addition to carrying information about the dimension of the index, tensor index objects can also be assigned "tags". These tags can be used to label the indices. On creation these tensor index objects are also assigned a permanent id number. In order to contract two tensor objects, their index id as well as tags should match for each site. As an example we can create a single tensor index with arbitray tags as follows

```
i = Index(2,"i,n=1,Site")
```

This creates an $Index$ type object of dimension 2 (i.e. it can have two values), with the tags "i", "$n = 1$", and "$Site$". The index id of this object can be found by $id(j)$.

Since we are interested in a spin chain of size N, we need N index objects. This task is handled by the function "siteinds". This takes as input N, site type, and information about conserved symmetries and their quantum number labels and constructs a vector of Tensor Index objects that have the relevant tags and properties. For our case if we are not using symmetries we can create the following "sites" vector

```
sites = siteinds("S=1/2",N)
```

This will create a vector of tensor index objects each of dimension 2, with tags "$S = 1/2$", "$Site$", and "$n = k$" where k varies from 1 to N for each site. ITensor has the several built-in site types. The physical tags of these are "S=1/2", "S=1", "Qubit", "Qudit", "Boson", "Fermion", "tJ", "Electron" [1]. We can also extend these site types by adding new operators or creating new site types [2] [3]. If we also want to include conserved quantum numbers corresponding to an abelian symmetry we can add the following keyword (separated by ; from the rest)

```
sites = siteinds("S=1/2",N; conserve_qns=true)
```

Some of the other built-in conserved symmetries are "conserve_sz", "conserve_szparity" etc [1] [4].

Also as mentioned previously tensor indices with the same id and tags can be contracted. Hence, on the lattice it is important to use the same tensor index vector "sites" above to construct all the MPO and MPS in our system. Having constructed the tensor indices we can now construct the Hamiltonian operator as a sum of operators using the function "OpSum", and then convert this to an MPO.

B. Operators and Operator Sum

There are several in-built local operators in ITensor for each of the site types mentioned previously [4] and new operators can also be constructed from their matrix form [2]. For the site type $S = 1/2$, the relevant preexisting operators are "Id", "Sz", "$S+$", and "$S-$". In order to use these strings as operators we need to associate them with tensor index with the appropriate site type tag. So for a single site tensor index object j with site type "$S = 1/2$", we can create an Sz operator as using the Op function of ITensor as follows

```
j = Index(2,"S=1/2")#This creates a single 2dim index of type S=1/2
Sz_op = op("Sz",j) #This creates a Sz operator acting on that site
```

For our requirement we want a combination of multiple operators acting at different sites of a lattice. To create a combination of operators with multiple terms we can use the $OpSum$ function of ITensor. Terms of the $OpSum$ object are a product of local operators along with their coefficients. Each term added to the $OpSum$ object has the general form $< coeff :: Number >, < Operator_1 :: String >< Index_1 :: Int >, < Operator_2 :: String >< Index_2 :: Int > ...$ This $OpSum$ object, along with the tensor index vector $sites$ created using $siteinds$ earlier, is used in the ITensor function MPO to create a Matrix Product Operator. For instance for the XY model the Hamiltonian is given as follows

$$H_{XY} = x \sum_{j=1}^{L-1} (\sigma_j^+ \sigma_{j+1}^- + \sigma_j^- \sigma_{j+1}^+) \tag{1}$$

To create and OpSum object, and consequently and MPO, from this the following code can be used

```
#creating the site index of type "S=1/2" for L sites
sites = siteinds("S=1/2",L)

#creating the OpSum
ampo = OpSum()
for j = 1:L-1
    ampo .+= x,"S+",j,"S-",j+1
    ampo .+= x,"S-",j,"S+",j+1
end

#creating the MPO
H_XY = MPO(ampo,sites) #sites created previously using siteinds
```

The MPO function assigns the OpSum object to indices created using siteinds which have the site type "$S = 1/2$".

We can similarly express the Coulomb part of the lattice Schwinger model Hamiltonian as H_C

$$H_C = y \cdot \sum_{j=1}^{L} \left[\sum_{k=1}^{j} \left(S_z + \frac{(-1)^k}{2} \right) + \frac{\theta}{2\pi} \right]^2 \tag{2}$$

In this for a fixed j, we can write the squared sum os

$$os = \sum_{k=1}^{j} \left(S_z(k) + \frac{(-1)^k}{2} \right) + \frac{\theta}{2\pi}, \tag{3}$$

then, to square this quantity, Julia does the multiplication by groups[1], i.e.

$$\begin{aligned}
os * os &= \left[\sum_{k=1}^{j}\left(S_z(k)+\frac{(-1)^k}{2}\right)+\frac{\theta}{2\pi}\right]^2 \\
&= \left[\sum_{k=1}^{j}\left(S_z(k)+\frac{(-1)^k}{2}\right)+\frac{\theta}{2\pi}\right]\left[\sum_{k=1}^{j}\left(S_z(k)+\frac{(-1)^k}{2}\right)+\frac{\theta}{2\pi}\right] \\
&= \left[\sum_{k=1}^{j}\left(S_z(k)+\frac{(-1)^k}{2}\right)+\frac{\theta}{2\pi}\right]\cdot S_z(1)-\frac{1}{2}\left[\sum_{k=1}^{j}\left(S_z(k)+\frac{(-1)^k}{2}\right)+\frac{\theta}{2\pi}\right]\cdot \mathbb{I}(1)\ +\cdots+ \\
&\quad +\left[\sum_{k=1}^{j}\left(S_z(k)+\frac{(-1)^k}{2}\right)+\frac{\theta}{2\pi}\right]\cdot S_z(j)+\frac{(-1)^j}{2}\left[\sum_{k=1}^{j}\left(S_z(k)+\frac{(-1)^k}{2}\right)+\frac{\theta}{2\pi}\right]\cdot \mathbb{I}(j)\ + \\
&\quad +\left[\sum_{k=1}^{j}\left(S_z(k)+\frac{(-1)^k}{2}\right)+\frac{\theta}{2\pi}\right]\cdot\frac{\theta}{2\pi} \\
&= \underbrace{os*os(1)}_{os\cdot S_z(1)}+\underbrace{os*os(2)}_{os\cdot\frac{(-1)}{2}\mathbb{I}(1)}+\cdots+\underbrace{os*os(2j-1)}_{os\cdot S_z(j)}+\underbrace{os*os(2j)}_{os\cdot\frac{(-1)^j}{2}\mathbb{I}(j)}+\underbrace{os*os(2j+1)}_{os\cdot\frac{\theta}{2\pi}}.
\end{aligned} \tag{4}$$

The code to implement this procedure is given below

```
ampo = OpSum()
for j in 1:L-1
    os = OpSum()
    for k in 1:j
        os .+= 1, "Sz", k
        os .+= 1/2*(-1)^k, "Id", k
    end
    os .+= theta/(2*pi), "Id", L
    # Square the previous sum for each j
    for k in 1:(2*j+1)
        ampo = y*os*os[k] + ampo
    end
end
# Add the last term (corresponding to L(N)=theta/(2*pi))
os += y*(theta/(2*pi))^2, "Id", L
```

Another way this same term can be implement is by expanding the square term as a product of $(j-1)^2$ terms and then summing over them. While logically more straightforward, this approach is much less efficient since it involves three nested loops which make the program slower. This approach can be implemented as

```
ampo = OpSum()
for j=1:L-1
    #for the Coulomb part
    for k = 1:j
        for l=1:j
            #ampo += y*(-1)^(k+l),"I",
            ampo .+= (y/4)*2*(-1)^k ,"Sz",l,"Id",k#Sz = sigma_z/2
            ampo .+= (y/4)*2*(-1)^l,"Id",l,"Sz",k
            ampo .+= (y/4)*4,"Sz",k,"Sz",l
            ampo .+= (y/4)*(-1)^(k+l),"Id",l,"Id",k
        end
    end
end
```

[1] Just writing $os * os$ doesn't give the explicit sum and prevents us to apply simple operator sums.

Similarly, an MPO for the mass term in the Hamiltonian can be constructed

$$H_{mass} = \mu \sum_{j=1}^{L} \left[\frac{1}{2} + (-1)^j S_z(j) \right]. \tag{5}$$

```
ampo = OpSum()
for j=1:L
    ampo .+= mu/2, "Id",j
    ampo .+= mu*(-1)^{j},"Sz",j
end
```

Therefore, for the massive Schwinger model with Hamiltonian given by

$$H = y \cdot \sum_{j=1}^{L-1} \left[\sum_{k=1}^{j} \left(S_z + \frac{(-1)^k}{2} \right) + \frac{\theta}{2\pi} \right]^2 + x \sum_{j=1}^{L-1} (\sigma_j^+ \sigma_{j+1}^- + \sigma_j^- \sigma_{j+1}^+) + \sum_{j=1}^{L} \left(\frac{1}{2} + (-1)^j S_z(j) \right) \tag{6}$$

The Hamiltonian MPO can be constructed as follows, assuming conserved symmetry

```
#site indices
sites = siteinds("S=1/2",L;conserve_qns=true)

#OpSum
ampo = OpSum()
for j=1:L-1

    #for the Coulomb part
    os = OpSum()
    for k in 1:j
        os .+= 1,"Sz", k
        os .+= 1/2*(-1)^k, "Id", k
    end
    os .+= theta/(2*pi), "Id", L
    # Square the previous sum for each j
    for k in 1:2*j+1
        ampo = y*os*os[k] + ampo
    end
    # Add the last term (corresponding to L(N)=theta/(2*pi))
    os += y*(theta/(2*pi))^2, "Id", L

    #For XY part
    ampo .+= x,"S+",j,"S-",j+1
    ampo .+= x,"S-",j,"S+",j+1

    #For mass term
    ampo .+= (-1)^(j)*mu,"Sz",j
    ampo .+= mu/2,"Id",j
end

#Mass term for the last site
ampo += (-1)^(L)*mu,"Sz",L;
ampo .+= mu/2,"Id",L

#constructing the MPO
H = MPO(ampo,sites)
```

III. MATRIX PRODUCT STATE

IV. DMRG

A. DMRG Command in Julia

Once formulated our Hamiltonian in an MPO form, we set the initial state to be consider as the initial state for the minimization procedure (such as Lanczos). This state must have the quantum numbers desired to be conserved, $S_z = 0$ in this case. We then formulate it in an MPS form and run the command

```
energy0, psi0 = dmrg(H, psi0_init; nsweeps, maxdim, cutoff,
                     noise, eigsolve_krylovdim, eigsolve_maxiter)
```

This command will return the ground energy $energy0$ and the ground state $psi0$ of the system. Where H is the Hamiltonian in an MPO form, $psi0\_init$ is the initial state in an MPS form, $nsweeps$ is the number of sweeps desired, $maxdim$ is the bond dimension of the operators (truncation m in DMRG), $cutoff$ is the maximum error allowed when sweeping, $noise$ to try to avoid local minima, $eigsolve\_krylovdim$ is the Krylov space dimension, and $eigsolve\_maxiter$ is the number of times the Krylov space can be rebuilt. An explicit example of a dmrg code in Julia is given below

```
# Create the MPO for the Hamiltonian
H = MPO(ampo, sites)

# Initialize the state with the quantum number (spin) desired
state = [isodd(n) ? "Up" : "Dn" for n in 1:N]

# Create an MPS for the previous state (with same spin if projection is used)
psi0_init = MPS(sites, state)

# Plan to do 5 DMRG sweeps:
nsweeps = 5

# Set maximum MPS bond dimensions for each sweep (truncation m)
maxdim = 8

# Set maximum error allowed when adapting bond dimensions
cutoff = [0]

# If DMRG is far from the global minumum then there is no guarantee
# that DMRG will be able to find the true ground state.
# This problem is exacerbated for quantum number conserving DMRG where
# the search space is more constrained.
# If this happens, a way out is to turn on the noise term feature to be
#a very small number.
noise = [1E-15]

# Maximum dimension of Krylov space to locally solve problem.
# Try setting to a higher value if convergence is slow or the Hamiltonian
# is close to a critical point.
eigsolve_krylovdim = 10
eigsolve_maxiter = 1    # Number of times Krylov space can be rebuild

# Run the DMRG algorithm, returning energy and optimized MPS of ground state
energy0, psi0 = dmrg(H, psi0_init; nsweeps, maxdim, cutoff, noise,
                     eigsolve_krylovdim, eigsolve_maxiter)
```

B. Excited States

To calculate the first excited state, the new Hamiltonian is defined as

$$H_1 = H + w \left|\psi_0\right\rangle \left\langle\psi_0\right| \tag{7}$$

where w is called *weight* and has to be at least bigger than the energy gap. Therefore, DMRG calculates the first state ψ_1 that minimizes H_1, i.e. minimizes the inner product $\langle\psi_0|\psi_1\rangle$, this is why we need w to be large enough so that $H_1 \left|\psi_1\right\rangle = H \left|\psi_1\right\rangle$. Thus, $\left|\psi_1\right\rangle$ is the first excited state of the system.

To obtain the nth excited state, we can generalize the previous procedure by defining the new Hamiltonian as

$$H_n = H + w_0 \left|\psi_0\right\rangle \left\langle\psi_0\right| + w_1 \left|\psi_1\right\rangle \left\langle\psi_1\right| + \ldots + w_{n-1} \left|\psi_{n-1}\right\rangle \left\langle\psi_{n-1}\right| \tag{8}$$

where the weights $w_0, \ldots w_{n-1}$ have to be large enough consistently so that the inner products $\langle\psi_0|\psi_n\rangle$, $\langle\psi_1|\psi_n\rangle$, $\ldots$, $\langle\psi_{n-1}|\psi_n\rangle$ are minimized. Therefore we can conclude that $H_n \left|\psi_n\right\rangle = H \left|\psi_n\right\rangle$, guaranteeing that $\left|\psi_n\right\rangle$ is the nth excited state of the system.

In order to calculate excited states using DMRG in Julia, we can use the following example code (this is a continuation of the previous code used to calculate the ground state):

```julia
x = 50
weight = 10*sqrt(x)

noise = [1E-11]

# Initialize the first excited state with same QNs as ground state
state1 = [if n>N/2 "Up" else "Dn" end for n in 1:N]
psi1_init = MPS(sites,state1)

# Run DMRG for energy and optimized MPS for first excited state
energy1,psi1 = dmrg(H,[psi0],psi1_init; nsweeps, maxdim, cutoff, noise, weight,
                    eigsolve_krylovdim, eigsolve_maxiter)

# Check if psi1 is orthogonal to psi0
@show inner(psi1,psi0)
```

where the *noise* term is explained in section VI and the last line is a check of the orthogonality between the ground and first excited states (check of accuracy).

V. RESULTS AND ANALYSIS

A. Chiral Condensate

The operator $\bar{\psi}\psi$ is a mass term, therefore on a lattice (this expression has to be shifted by $-L/2$ in order to match the exact results, namely, to have vacuum energy of $-L\mu/2$):

$$\bar{\psi}\psi/e \to \frac{\sqrt{x}}{L} \sum_n (-1)^n \phi^\dagger(n)\phi(n) = \frac{\sqrt{x}}{L} \sum_n \frac{1}{2}\left[1 + (-1)^n \sigma_z(n)\right] = \frac{\sqrt{x}}{L} \sum_n \left[\frac{1}{2} + (-1)^n S_z(n)\right]. \tag{9}$$

Therefore, the condensate value is

$$\left\langle\bar{\psi}\psi\right\rangle/e = \frac{\sqrt{x}}{L} \left\langle\Psi_0\right| \sum_{n=1}^{L} \left[\frac{1}{2} + (-1)^n S_z(n)\right] \left|\Psi_0\right\rangle, \tag{10}$$

where $\left|\Psi_0\right\rangle$ is the ground state.

In order to evaluate this we first create an MPO object "*ccmpo*" that is associated with the same site indices as the ground state MPS "*psi*0" we got from dmrg.

```
#Constructing the OpSum
ccampo = OpSum()
for j=1:L
    ccampo .+= (-1)^(j),"Sz",j
    ccampo .+= 1/2,"Id",j
end
#Constructing the MPO
ccmpo = MPO(ccampo,sites)#same "sites" as the one used for H

#Evaluating the expectation value
ChiralCondesate = inner(psi0',ccmpo,psi0)*sqrt(x)/L
```

Where we have evaluated the expectation value of *ccmpo* between $psi0'$ and $psi0$ and multiplied by $\frac{\sqrt{x}}{L}$ to get the Chiral Condensate per unit coupling.

B. Order parameter

When we consider the massive Schwinger model with a non-zero background electric field of $F = \pm e/2$ or, equivalently $\theta = \pm\pi$, we have a phase transition. The reason is that there is a spontaneous symmetry breaking of the ground state symmetry $\mathbb{Z}_2$ when going from strong coupling $m/e \ll 1$ to weak coupling $m/e \gg 1$. In order to test this, the obvious order parameter to use is the electric field expectation value $\langle E \rangle$, or similarly (by bosonization) the boson formed by fundamental fermions expectation value $\langle \phi \rangle$. Now, as we are interested when it goes from zero to a non-zero value, we can instead use $\langle \sin\phi \rangle$. The reason is because this expresion is equivalent (using bosonization) to:

$$\begin{aligned}\langle i\bar{\psi}\gamma^5\psi\rangle /e &\to \frac{i\sqrt{x}}{L}\langle\Psi_0|\sum_{n=1}^{L-1}(-1)^n\left[\phi^\dagger(n)\phi(n+1)-\phi^\dagger(n+1)\phi(n)\right]|\Psi_0\rangle \\ &= \frac{\sqrt{x}}{L}\langle\Psi_0|\sum_{n=1}^{L-1}(-1)^n\left[\sigma^+(n)\sigma^-(n+1)+\sigma^+(n+1)\sigma^-(n)\right]|\Psi_0\rangle\,. \end{aligned} \tag{11}$$

From this, we can extract the critical value μ_c that corresponds to the point in which the transition occurs and compare it with the result of $(m/e)_c \approx 0.3335$.

C. Mass Gap

The Schwinger mass is $M = \frac{e}{\sqrt{\pi}}$. On a lattice of size L, this can be computed by computing the energy difference between the ground state and the first excited state.

$$M = \frac{E_1 - E_0}{2\sqrt{x}} \tag{12}$$

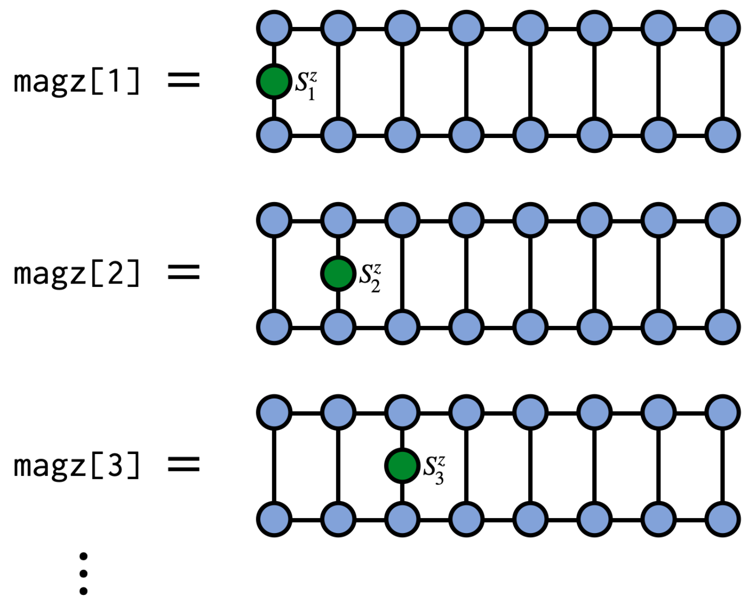
Where $x = 1/(ea)^2$ and E_1,E_0 can be computed using the dmrg function.

D. Expectation Values and Correlation Matrix

If we are interested in computing $\langle\Psi|O_j|\Psi\rangle$, the expectation value of a local operator O_j acting on a particular site j in an MPS state $|\Psi\rangle$, the *expect* function can be used [5]. For instance to compute the expectation value of "S_z" operator acting at a particular site in the MPS of the ground state $psi0$ computed from dmrg, we have

```
magz = expect(psi0,"Sz")
#magz[j] = <psi0|Sz(j)|psi0>
```

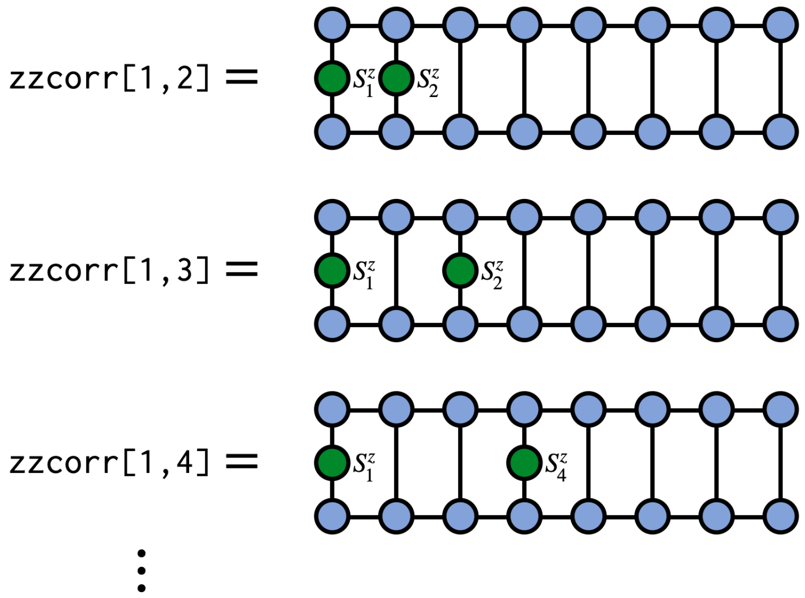
This gives a vector of same length as the number of sites in the MPS, and whose j^{th} element is $\langle\Psi_0|S_z(j)|\Psi_0\rangle$. Pictorially, the elements of this vector can be represented as [5]



If one is interested in computing the correlation matrix $C_{ij} = \langle\Psi|A_i B_j|\Psi\rangle$, for local operators A and B acting on sites i and j respectively we use the *correlation_matrix* function [5]. As an example the correlation matrix for two S_z operators in the ground state *psi*0 is computed by

```
zzcorr = correlation_matrix(psi0,"Sz","Sz")
```

As the name suggests, this creates a matrix with $zzcorr[i,j] = \langle\Psi|A_i B_j|\Psi\rangle$. The elements of this matrix are pictorially represented as [5]



On the other hand, if we are interested in finding the expectation value of a global operator P with respect to the MPS state $|\psi\rangle$, namely the value of $\langle\psi|\,O\,|\psi\rangle$, then we use the command *inner*, as follows:

```
inner(psi’,P,psi),
```

where psi’ represents the dagger of the state psi= $|\psi\rangle$ that is psi’= $\langle\psi|$.

E. Entanglement Entropy of MPS

Since the entanglement is closely tied with the concept of MPS and MPO, it should not be unexpected that calculating the entanglement entropy between two parts of an MPS is not difficult. Consider a bipartition of an MPS *psi* into region "A" and "B" consisting of sites $1, 2, ..., r$ and $r + 1, ..., L$. Then using the singular value decomposition allows for the computation of the entanglement entropy.

```
#Shifting the orthogonality centre to site r
#Tensors left/right of the orthogonality centre are orthogonal
orthogonalize!(psi, r)

#performing singular value decomposition
U,S,V = svd(psi[r], (linkind(psi, r-1), siteind(psi,r)))

SvN = 0.0 #initializing the entanglement entropy
for n=1:dim(S, 1)
  p = S[n,n]^2 #square of the diagonal elements give p_n
  SvN -= p * log(p)
end
```

The SvN above gives the Von Neumann entanglement entropy between the two partitions.

F. Saving Results

Calculation of the above quantities mainly requires knowledge about the state of the system. For large lattice size and truncation dimension, computation of the relevant MPS from the dmrg calculations can take a lot of time and computing power. Therefore, it would be prudent to save these results. ITensor allows for saving of the MPS and MPO in an HDF5 file format [5] [6]. To save an MPS *psi* with the name "psi" in the file *filename.h5* we use the following

```
#To save the MPS
using ITensors.HDF5
f = h5open("filename.h5","w")
write(f,"psi",psi)
close(f)
```

This will save the ".h5" file in the current directory. Having saved the MPS, in order to retrieve use the following

```
#To read a saved MPS
using ITensors.HDF5
f = h5open("filename.h5","r") #"r" instead of "w"
psi = read(f,"psi",MPS) #using read function with MPS as an argument
close(f)
```

As we saw earlier, in order to use these MPS we would also need the site indices associated with these. This can be recovered by

```
sites = siteinds(psi)
```

Now, these site indices can be used to construct MPO's and find their expectation values.

VI. CONVERGENCE

In principle, there is no way to know if DMRG has converged. So, if we run into the case where DMRG is far from the global minumum then there is no guarantee that DMRG will be able to find the true ground state. This problem is even worse when we use projection because the Hilbert space is constrained. In this case, a simple way out is to turn on the *noise* term and let it be a very small number. A coding example is:

```
# Create the MPO for the Hamiltonian
H = MPO(ampo, sites)
```

```
# Initialize the state with the quantum number (spin) desired
state = [isodd(n) ? "Up" : "Dn" for n in 1:N]

# Create an MPS for the previous state (with same spin if projection is used)
psi0_init = MPS(sites, state)

# Plan to do 5 DMRG sweeps:
nsweeps = 5

# Set maximum MPS bond dimensions for each sweep (truncation m)
maxdim = 8

# Set maximum error allowed when adapting bond dimensions
cutoff = [1E-13]

#Set the noise term
noise = [1E-11]

# Run the DMRG algorithm, returning energy and optimized MPS of ground state
energy0, psi0 = dmrg(H, psi0_init; nsweeps, maxdim, cutoff, noise,
                     eigsolve_krylovdim, eigsolve_maxiter)
```

VII. USING JULIA IN SLURM

First, we need to download *conda* in our workspace, the steps are[2]:

1. Download Miniconda running the command: "wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh"
2. Make the installation script executable by running the command: "chmod +x Miniconda3-latest-Linux-x86_64.sh"
3. Run the installation script with the command: "./Miniconda3-latest-Linux-x86_64.sh" and follow the prompts to complete the installation.
4. If we want to use conda on SLURM cluster, we should add the path to the .bashrc file by running the command: "echo 'export PATH="/path/to/miniconda3/bin:$PATH"' » ∼ /.bashrc"
5. Finally, activate the changes by running the command "source ∼/.bashrc".

Now, we install *Julia*, by typing: "conda install -c conda-forge julia". We can check if Julia was installed by opening it in our workspace with the command "julia". Finally, the command to open a script in a batch for a Julia job is simply "julia *name-of-script.jl*".

[1] "SiteType and op, state, val functions xb7; ITensors.jl — itensor.github.io." `https://itensor.github.io/ITensors.jl/dev/SiteType.html`. [Accessed 28-Apr-2023].
[2] "Physics (SiteType) System Examples xb7; ITensors.jl — itensor.github.io." `https://itensor.github.io/ITensors.jl/stable/examples/Physics.html`. [Accessed 28-Apr-2023].
[3] "Physics System Examples xB7; ITensors.jl — docs.juliahub.com." `https://docs.juliahub.com/ITensors/P3pqL/0.2.0/examples/Physics.html`. [Accessed 28-Apr-2023].
[4] "SiteTypes Included with ITensor xb7; ITensors.jl — itensor.github.io." `https://itensor.github.io/ITensors.jl/dev/IncludedSiteTypes.html`. [Accessed 28-Apr-2023].

[2] Taken from https://samanemami.medium.com/how-to-install-miniconda-on-linux-using-slurm-step-by-step-99fd2043c7c2

[5] "MPS and MPO Examples xb7; ITensors.jl — itensor.github.io." `https://itensor.github.io/ITensors.jl/dev/examples/MPSandMPO.html`. [Accessed 03-May-2023].

[6] "HDF5 File Formats xb7; ITensors.jl — itensor.github.io." `https://itensor.github.io/ITensors.jl/dev/HDF5FileFormats.html#mpo_hdf5`. [Accessed 05-May-2023].