ITensor

E.M. Stoudenmire
Simons Summer School - June 2016
ITensor – Key features

• Tensor indices carry extra info and matching indices automatically contract

• Real/complex and dense/sparse/block-sparse tensors

• Useful not only for MPS/DMRG but general tensor network algorithms

• DMRG codes with many features (excited states; sums of Hamiltonians); combine other algs. with DMRG

• AutoMPO for making MPOs from code resembling hand written mathematical notation
ITensor—Intelligent Tensor—is a C++ library for implementing tensor product wavefunction calculations. It is efficient and flexible enough to be used for research-grade simulations.

Features include:
- Named indices; no need to think about index ordering
- Full-featured matrix product state and DMRG layer
- Quantum number conserving (block-sparse) tensors; same interface as dense tensors
- Complex numbers handled lazily: no efficiency loss if real
- Easy to install; only dependencies are BLAS/LAPACK and C++11

ITensors have an Einstein summation interface making them nearly as easy to multiply as scalars: tensors indices have unique identities and matching indices automatically contract when two ITensors are multiplied. This type of interface makes it simple to transcribe tensor network diagrams into correct, efficient code.

For example, the diagram below (resembling the overlap of matrix product states) can be converted to code as

\[
\begin{align*}
A \otimes C & \quad = \quad A \otimes B \otimes C \otimes D
\end{align*}
\]
DMRG With ITensor
ITensor DMRG code  (provided in sample/dmrg.cc)

```cpp
int N = 100;
auto sites = SpinOne(N);

auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5, "S+", j, "S-", j+1;
    ampo += 0.5, "S-", j, "S+", j+1;
    ampo += "Sz", j, "Sz", j+1;
}
auto H = MPO(ampo);

auto psi = MPS(sites);

Sweeps sweeps(5);
sweeps.maxm() = 10, 20, 100, 100, 200;

dmrg(psi, H, sweeps, "Quiet");
```
Itensor DMRG code

```cpp
int N = 100;
auto sites = SpinOne(N);
```
ITensor DMRG code

```c++
int N = 100;
auto sites = SpinOne(N);
```
int N = 100;
auto sites = SpinOne(N);

auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5, "S+", j, "S-", j+1;
    ampo += 0.5, "S-", j, "S+", j+1;
    ampo += "Sz", j, "Sz", j+1;
}
auto H = MPO(ampo);
```cpp
int N = 100;
auto sites = SpinOne(N);

auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5, "S+", j, "S-", j+1;
    ampo += 0.5, "S-", j, "S+", j+1;
    ampo += "Sz", j, "Sz", j+1;
}
auto H = MPO(ampo);
```
int N = 100;
auto sites = SpinOne(N);

auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j) {
    ampo += 0.5, "S+", j, "S-", j+1;
    ampo += 0.5, "S-", j, "S+", j+1;
    ampo += "Sz", j, "Sz", j+1;
}
auto H = MPO(ampo);

$H = \sum_j \frac{1}{2} S_j^+ S_{j+1}^- + \frac{1}{2} S_j^- S_{j+1}^+ + S_j^z S_{j+1}^z$
int N = 100;
auto sites = SpinOne(N);

auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5, "S+", j, "S-", j+1;
    ampo += 0.5, "S-", j, "S+", j+1;
    ampo += "Sz", j, "Sz", j+1;
}

auto H = MPO(ampo);

auto psi = MPS(sites);
auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5,"S+",j,"S-",j+1;
    ampo += 0.5,"S-",j,"S+",j+1;
    ampo += "Sz",j,"Sz",j+1;
}
auto H = MPO(ampo);

auto psi = MPS(sites);

Sweeps sweeps(5);
sweeps.maxm() = 10,20,100,100,200;

dmrg(psi,H,sweeps,"Quiet");
auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5,"S+",j,"S-",j+1;
    ampo += 0.5,"S-",j,"S+",j+1;
    ampo += "Sz",j,"Sz",j+1;
}
auto H = MPO(ampo);

auto psi = MPS(sites);

Sweeps sweeps(5);
sweeps.maxm() = 10,20,100,100,200;

dmrg(psi,H,sweeps,"Quiet");
Hands on practice:

1. Read then run the `dmrg.cc` sample code (100 site S=1 Heisenberg chain)
   - `cd sample/` inside ITensor folder
   - `make dmrg`
   - `./dmrg`

2. Change the size to N=110 sites, re-run

3. Compute $E = (E_{110} - E_{100})/10$ [the "subtraction trick"]
   compare to exact energy density

chain. The density-matrix renormalization-group techniques used allow us to calculate a variety of properties of the chain with unprecedented accuracy. The ground state energy per site of the infinite chain is found to be $e_0 \equiv -1.401484038971(4)$. Open-ended $S = 1$ chains have effective
Extending DMRG
auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5, "S+", j, "S-", j+1;
    ampo += 0.5, "S-", j, "S+", j+1;
    ampo += "Sz", j, "Sz", j+1;
}
auto H = MPO(ampo);

\[ H = \sum_{j} \frac{1}{2} S_{j}^{+} S_{j+1}^{-} + \frac{1}{2} S_{j}^{-} S_{j+1}^{+} + S_{j}^{z} S_{j+1}^{z} \]
auto Jz = 0.8;

auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5,"S+",j,"S-",j+1;
    ampo += 0.5,"S-",j,"S+",j+1;
    ampo += Jz,"Sz",j,"Sz",j+1;
}
auto H = MPO(ampo);

\[
H = \sum_j \frac{1}{2} S_j^+ S_{j+1}^- + \frac{1}{2} S_j^- S_{j+1}^+ + J_z S_j^z S_{j+1}^z
\]
auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5,"S+",j,"S-",j+1;
    ampo += 0.5,"S-",j,"S+",j+1;
    ampo += "Sz",j,"Sz",j+1;
}

auto H = MPO(ampo);

\[ H = \sum_{j} \vec{S}_j \cdot \vec{S}_{j+1} \]
Modify Terms in Hamiltonian

```cpp
auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5,"S+",j,"S-",j+1;
    ampo += 0.5,"S-",j,"S+",j+1;
    ampo += "Sz",j,"Sz",j+1;
}
for(int j = 1; j < N-1; ++j)
{
    ampo += 0.5*J2,"S+",j,"S-",j+2;
    ampo += 0.5*J2,"S-",j,"S+",j+2;
    ampo += J2,"Sz",j,"Sz",j+2;
}
auto H = MPO(ampo);
```

\[
H = \sum_j \vec{S}_j \cdot \vec{S}_{j+1} + J_2 \sum_j \vec{S}_j \cdot \vec{S}_{j+2}
\]
auto sites = SpinOne(N);

auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5,"S+",j,"S-",j+1;
    ampo += 0.5,"S-",j,"S+",j+1;
    ampo += "Sz",j,"Sz",j+1;
}
auto H = MPO(ampo);

\[ H = \sum_j \vec{S}_j \cdot \vec{S}_{j+1} \]
auto sites = SpinHalf(N);

auto ampo = AutoMPO(sites);
for(int j = 1; j < N; ++j)
{
    ampo += 0.5,"S+",j,"S-",j+1;
    ampo += 0.5,"S-",j,"S+",j+1;
    ampo += "Sz",j,"Sz",j+1;
}
auto H = MPO(ampo);

\[ H = \sum_j \vec{S}_j \cdot \vec{S}_{j+1} \]
auto sites = Spinless(N);

auto ampo = AutoMPO(sites);
for(int j = 1; j <= N-1; ++j)
{
    ampo += -t/2.,"Cdag",j, "C",j+1;
    ampo += -t/2.,"Cdag",j+1,"C",j;
}
for(int j = 1; j <= N; ++j)
{
    ampo += (V(j)-mu),"N",j;
}
auto H = MPO(ampo);

\[ H = -\frac{t}{2} \sum_j c_j^\dagger c_{j+1} + c_{j+1}^\dagger c_j + \sum_j (V_j - \mu) n_j \]
auto lattice = squareLattice(Nx,Ny,{"YPeriodic",true});

auto ampo = AutoMPO(sites);
for(auto b : lattice)
{
    ampo += 0.5,"S+",b.s1,"S-",b.s2;
    ampo += 0.5,"S-",b.s1,"S+",b.s2;
    ampo += "Sz",b.s1,"Sz",b.s2;
}
auto H = MPO(ampo);
ITensor Tutorial
Consider a single-site wavefunction, for example a spin 1/2

Single-site basis:

$$|s = 1\rangle = |\uparrow\rangle$$

$$|s = 2\rangle = |\downarrow\rangle$$
Most general wavefunction for a spin 1/2:

$$|\psi\rangle = \sum_{s=1}^{2} \psi_s |s\rangle$$

The $\psi_s$ are complex numbers.

Can view $\psi_s$ as a tensor (one index):
Single-site wavefunction as a tensor:

\[ \psi_s \rightarrow s \]

Using ITensor:

```cpp
auto s = Index("index s", 2);
auto psi = ITensor(s);
```
Now initialize $\psi_s$  First choose $|\psi\rangle = |\uparrow\rangle$

\[
1 \begin{array}{c}
\bullet
\end{array} = 1
\]

```cpp
auto s = Index("s",2);
auto psi = ITensor(s);
psi.set(s(1), 1.0);
PrintData(psi);
```
Now initialize $\psi_s$  
First choose $|\psi\rangle = |\uparrow\rangle$

```
auto s = Index("s",2);
auto psi = ITensor(s);
psi.set(s(1), 1.0);
PrintData(psi);
```

```
psi = ITensor r=1: (s,2,Link,273)
(1) 1.00
```
Make some operators:

```cpp
class ITensor {
public:
    ITensor(auto s, auto prime_s) 

    auto Sz = ITensor(s, prime(s));
    auto Sx = ITensor(s, prime(s));
}
```

New ITensors start out set to zero

What does “prime” do?

`prime(s)` returns copy of `s` with a “prime level” of 1

Could use different indices (say `s` and `t`), but `s'` more convenient - can remove prime later
Our operators:

```cpp
auto Sz = ITensor(s,prime(s));
auto Sx = ITensor(s,prime(s));
```

Set their components:

```cpp
Sz.set(s(1),prime(s)(1), +0.5);
Sz.set(s(2),prime(s)(2), -0.5);
Sx.set(s(1),prime(s)(2), +0.5);
Sx.set(s(2),prime(s)(1), +0.5);
```
Let’s compute $\hat{S}_x |\psi\rangle = |\phi\rangle$

$$(\hat{S}_x)_{s'}^s |\psi_s\rangle = \begin{array}{c} \text{green node} \\ s' \end{array} = \begin{array}{c} \text{blue node} \\ s \end{array} = \begin{array}{c} \text{purple node} \\ s' \end{array}$$

In code,

```c++
ITensor phi = Sx * psi;
```

The $*$ operator *contracts all matching indices*

Indices $s$ and $s'$ don’t match because of their different prime levels
What state is $\phi$?

$$(\hat{S}_x)_{s'}^s \psi_s = \begin{array}{c} \text{ITensor } \phi \text{ = } Sx \ast \psi; \\ \text{PrintData}(\phi); \end{array}$$
What state is $\phi$?

$$(\hat{S}_x)_{s'}^s \psi_s = \begin{array}{c}
\text{ITensor } \phi = Sx \ast \psi;
\text{PrintData}(\phi);
\phi = \begin{array}{c}
\text{ITensor } r=1: (s,2,\text{Link},273)'
\end{array}
\end{array}$$

\begin{array}{c}
(2) 0.500
\end{array}$$
More interesting $\psi_s$: choose $\theta = \pi/4$ and

\[
\begin{align*}
1 & = \cos \theta/2 \\
2 & = \sin \theta/2
\end{align*}
\]

Real theta = Pi/4.;

psi.set(s(1),cos(theta/2));
psi.set(s(2),sin(theta/2));

PrintData(psi);
More interesting $\psi_s$: choose $\theta = \pi/4$ and

\[
\begin{align*}
1 & = \cos \theta / 2 \\
2 & = \sin \theta / 2
\end{align*}
\]

Real theta = Pi/4.;

```java
psi.set(s(1), cos(theta/2));
psi.set(s(2), sin(theta/2));
PrintData(psi);
```

psi = ITensor r=1: (s,2,Link,273)
(1) 0.92388
(2) 0.38268
Diagrammatically, measurements (expectation values) look like:

For convenience, make:

```
ITensor cpsi = dag(prime(psi));
```

Calculate expectation values:

```
auto zz = (cpsi * Sz * psi).real();
auto xx = (cpsi * Sx * psi).real();
```
auto zz = (cpsi * Sz * psi).real();
auto xx = (cpsi * Sx * psi).real();

Printing the results,

println("<Sz> = ", zz);
println("<Sx> = ", xx);

we get the output

<Sz> = 0.35355
<Sx> = 0.35355

\[
\sqrt{(0.35355)^2 + (0.35355)^2} = 1/2
\]
Review:

- Construct an Index
  ```cpp
  auto a = Index("index a", 4);
  ```

- Construct ITensor (indices a, b, c)
  ```cpp
  auto T = ITensor(a, b, c);
  ```

- Set ITensor components
  ```cpp
  T.set(a(2), c(3), b(1), 7.89);
  ```

- Prime an Index b → b'
  ```cpp
  prime(b)
  ```

- The * operator automatically contracts matching Index pairs
Quiz:

If we * the following tensors, how many indices remain?
Quiz:

If we * the following tensors, how many indices remain?
Quiz:

If we * the following tensors, how many indices remain?
Code hands-on — in your ITensor folder
   cd tutorial/01_one_site

1. Read the code, then compile by typing “make”
   Run by typing “./one”

2. Change psi to be an eigenstate of $S_x$
   \[
   |\psi\rangle = \frac{1}{\sqrt{2}} (|\uparrow\rangle + |\downarrow\rangle)
   \]

3. Compute overlap of \(|\psi\rangle\) with \(|\phi\rangle = \hat{S}_x |\psi\rangle\)
   ```
   auto phi = Sx * psi;
   phi.noprime();
   auto olap = (dag(psi)*phi).real();
   ```

4. Try normalizing phi before computing overlap
   ```
   phi /= norm(phi);
   ```
Most general two-site wavefunction is

$$\left| \Psi \right\rangle = \sum_{s_1, s_2 = 1}^{2} \psi_{s_1 s_2} \left| s_1 \right\rangle \left| s_2 \right\rangle$$

Amplitudes are a rank-2 tensor

$$\psi_{s_1 s_2} =$$
Let’s make a singlet

\[ \frac{1}{\sqrt{2}} \]

\[ \frac{-1}{\sqrt{2}} \]

**Using ITensor:**

```cpp
auto s1 = Index("s1",2);
auto s2 = Index("s2",2);

auto psi = ITensor(s1,s2);

psi.set(s1(1),s2(2), +1./sqrt(2));
psi.set(s1(2),s2(1), -1./sqrt(2));
```
Interesting ITensor fact: no dependence on Index order:

```plaintext
psi.set(s1(1), s2(2), +1./sqrt(2));
psi.set(s1(2), s2(1), -1./sqrt(2));
```

Same result as:

```plaintext
psi.set(s1(1), s2(2), +1./sqrt(2));
psi.set(s2(1), s1(2), -1./sqrt(2));
```
Let’s make the Heisenberg Hamiltonian \( \hat{H} = S_1 \cdot S_2 \)

\[
\hat{H} = S_1^z S_2^z + \frac{1}{2} S_1^+ S_2^- + \frac{1}{2} S_1^- S_2^+
\]

First create operators, for example \( S^+ \)

```cpp
auto Sp1 = ITensor(s1,prime(s1));
Sp1.set(s1(2),prime(s1)(1), 1);
```

Multiply and add operators to make \( H \):

```cpp
auto H = Sz1*Sz2 + 0.5*Sp1*Sm2 + 0.5*Sm1*Sp2;
```
Tensor form of $H$

$$\hat{H} = \begin{align*}
\hat{H} &= \quad \quad + \quad \frac{1}{2} \quad + \quad \frac{1}{2} \\
= &
\end{align*}$$

Showing Index labels

$$\hat{H} = \begin{pmatrix}
\hat{H} & s'_{1} & s'_{2} \\
\hat{H} & s_{1} & s_{2}
\end{pmatrix}$$

```python
auto H = Sz1*Sz2 + 0.5*Sp1*Sm2 + 0.5*Sm1*Sp2;
```
Compute singlet energy with this Hamiltonian:

\[ \hat{H} |\psi\rangle = \hat{H} \psi = \psi \]

\[
\hat{H} = \begin{pmatrix}
    s_1' & s_2' \\
    s_1 & s_2
\end{pmatrix}
\]

auto Hpsi = H * psi;
Hpsi.noprime();

Real E = (dag(Hpsi) * psi).real();
Print(E);
//prints: E = -0.75
Compute singlet energy with this Hamiltonian:

\[ \hat{H} \ket{\psi} = \begin{pmatrix} s_1' & s_2' \\ s_1 & s_2 \end{pmatrix} \hat{H} \begin{pmatrix} s_1' & s_2' \\ s_1 & s_2 \end{pmatrix} \hat{H} \psi = \hat{H} \psi; \quad E = \begin{pmatrix} s_1 & s_2 \end{pmatrix} \hat{H} \psi \]

auto Hpsi = H * psi;
Hpsi.noprime();

Real E = (dag(Hpsi) * psi).real();
Print(E);
//prints: E = -0.75
Or compute energy in one shot:

\[ E_{\text{Sing}} = \text{dag}(\text{prime}(\psi)) \]

\[
\begin{align*}
\text{Real } E &= (\text{dag}(\text{prime}(\psi)) \ast H \ast \psi).\text{real}(); \\
\text{Print}(E); \\
//&\text{prints: } E = -0.75
\end{align*}
\]
For an arbitrary Hamiltonian, can find ground state by doing imaginary time evolution

\[ e^{-\beta H} |\Psi_{\text{init}}\rangle \propto |\Psi_0\rangle \]

```cpp
auto beta = 10.;
auto expH = expHermitian(H,-beta);

auto psibeta = expH * psi;
psibeta.noprime();
```
For an arbitrary Hamiltonian, can find ground state by doing imaginary time evolution

\[ e^{-\beta H} |\Psi_{\text{init}}\rangle \propto |\Psi_0\rangle \]

\[
\begin{array}{c}
\text{auto beta = 10.;} \\
\text{auto expH = expHermitian(H,-beta);} \\
\text{auto psibeta = expH * psi;} \\
\text{psibeta.noprime();}
\end{array}
\]

\[ e^{-\beta H} \Psi = \Psi_\beta \]
The density matrix renormalization group (DMRG) uses a variational wavefunction known as a matrix product state (MPS).

Matrix product states arise from compressing a one-dimensional wavefunction using the singular-value decomposition (SVD).

Let’s see how this works...
Recall:
Singular-value decomposition

Given rectangular (4x3) matrix \( M \)

\[
M = \begin{bmatrix}
0.435839 & 0.223707 & 0.10 \\
0.435839 & 0.223707 & -0.10 \\
0.223707 & 0.435839 & 0.10 \\
0.223707 & 0.435839 & -0.10
\end{bmatrix}
\]

Can decompose as

\[
\begin{bmatrix}
1/2 & -1/2 & 1/2 \\
1/2 & -1/2 & -1/2 \\
1/2 & 1/2 & 1/2 \\
1/2 & 1/2 & -1/2
\end{bmatrix}
\begin{bmatrix}
0.933 & 0 & 0 \\
0 & 0.300 & 0 \\
0 & 0 & 0.200
\end{bmatrix}
\begin{bmatrix}
0.707107 & 0.707107 & 0 \\
-0.707107 & 0.707107 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
Matrices $A$ and $B$ are "isometries":

\[ A^\dagger A = 1 \]
\[ BB^\dagger = 1 \]

$D$ diagonal

Elements of $D$ can be chosen:

1. Real
2. Positive semi-definite
3. Decreasing order
Keep fewer and fewer elements of \(D\):

\[
A = \begin{bmatrix}
\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\
\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & \frac{1}{2} & -\frac{1}{2}
\end{bmatrix}
\quad D = \begin{bmatrix}
0.933 & 0 & 0 \\
0 & 0.300 & 0 \\
0 & 0 & 0.200
\end{bmatrix}
\quad B = \begin{bmatrix}
0.707107 & 0.707107 & 0 \\
-0.707107 & 0.707107 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
M = \begin{bmatrix}
0.435839 & 0.223707 & -0.10 \\
0.435839 & 0.223707 & -0.10 \\
0.223707 & 0.435839 & 0.10 \\
0.223707 & 0.435839 & -0.10
\end{bmatrix}
\]

\[
\| M - M \|^2 = 0
\]
Keep fewer and fewer elements of \( D \):

\[
\begin{bmatrix}
\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\
\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & \frac{1}{2} & -\frac{1}{2}
\end{bmatrix}
\begin{bmatrix}
0.933 & 0 & 0 \\
0 & 0.300 & 0 \\
0 & 0 & 0.200
\end{bmatrix}
\begin{bmatrix}
0.707107 & 0.707107 & 0 \\
-0.707107 & 0.707107 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
= M_2 = 
\begin{bmatrix}
0.435839 & 0.223707 & 0 \\
0.435839 & 0.223707 & 0 \\
0.223707 & 0.435839 & 0 \\
0.223707 & 0.435839 & 0
\end{bmatrix}
\]

\[
\| M - M_2 \|^2 = 0
\]
Keep fewer and fewer elements of $D$:

$$
\begin{bmatrix}
1/2 & -1/2 & 1/2 \\
1/2 & -1/2 & -1/2 \\
1/2 & 1/2 & 1/2 \\
1/2 & 1/2 & -1/2 \\
\end{bmatrix}
\begin{bmatrix}
0.933 & 0 & 0 \\
0 & 0.300 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
0.707107 & 0.707107 & 0 \\
-0.707107 & 0.707107 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
0.435839 & 0.223707 & 0 \\
0.435839 & 0.223707 & 0 \\
0.223707 & 0.435839 & 0 \\
0.223707 & 0.435839 & 0 \\
\end{bmatrix}
= M_2 =
\begin{bmatrix}
0.435839 & 0.223707 & 0 \\
0.435839 & 0.223707 & 0 \\
0.223707 & 0.435839 & 0 \\
0.223707 & 0.435839 & 0 \\
\end{bmatrix}
$$

$$||M_2 - M||^2 = 0.04 = (0.2)^2$$
Keep fewer and fewer elements of $D$:

$$
\begin{bmatrix}
1/2 & -1/2 & 1/2 \\
1/2 & -1/2 & -1/2 \\
1/2 & 1/2 & 1/2 \\
1/2 & 1/2 & -1/2 \\
\end{bmatrix}
\begin{bmatrix}
0.933 & 0 & 0 \\
0 & 0.300 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
0.707107 & 0.707107 & 0 \\
-0.707107 & 0.707107 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
M_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
0.329773 & 0.329773 & 0 \\
0.329773 & 0.329773 & 0 \\
0.329773 & 0.329773 & 0 \\
0.329773 & 0.329773 & 0 \\
\end{bmatrix}

$$

$$
\|M_2 - M\|^2 = 0.04 = (0.2)^2
$$
Keep fewer and fewer elements of D:

$$A = \begin{bmatrix} 1/2 & -1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 \end{bmatrix}, \quad D = \begin{bmatrix} 0.933 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0.707107 & 0.707107 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0.707107 & 0.707107 & 0 \\ -0.707107 & 0.707107 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= M_3 = \begin{bmatrix} 0.329773 & 0.329773 & 0 \\ 0.329773 & 0.329773 & 0 \\ 0.329773 & 0.329773 & 0 \\ 0.329773 & 0.329773 & 0 \end{bmatrix}$$

$$\|M_3 - M\|^2 = 0.13 = (0.3)^2 + (0.2)^2$$
Keep fewer and fewer elements of $D$:

$$A = \begin{bmatrix} 1/2 & -1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 \end{bmatrix} \quad D = \begin{bmatrix} 0.933 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0.707107 & 0.707107 & 0 \\ -0.707107 & 0.707107 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= M_3 = \begin{bmatrix} 0.329773 & 0.329773 & 0 \\ 0.329773 & 0.329773 & 0 \\ 0.329773 & 0.329773 & 0 \\ 0.329773 & 0.329773 & 0 \end{bmatrix}$$

Truncating SVD = Controlled approximation for $M$

$$\|M_3 - M\|^2 = 0.13 = (0.3)^2 + (0.2)^2$$
Recall:

Most general two-spin wavefunction

\[ \psi_{s_1 s_2} = \]

Can treat as a matrix:

\[ \psi_{s_1 s_2} = \]

\[ s_1 \quad s_2 \]

\[ s_1 \quad s_2 \]
SVD this matrix:

\[ \psi_{s_1 s_2} = s_1 \quad s_2 \]

Bend lines back to look like wavefunction:
Using ITensor:

Say we have a two-site wavefunction $\psi$

 Declare $A,D,B$ to hold results of SVD

```cpp
auto A = ITensor(s1)
ITensor D,B;
```

Call SVD function

```cpp
svd(psi,A,D,B);
```
What have we gained from SVD?

Generic two-spin wavefunction (say spin $S$):

$\begin{bmatrix} s_1 \\ s_2 \end{bmatrix}$

$(2S+1)^2$ parameters

Not clear which parameters important, unimportant

Compressed wavefunction:

$\begin{bmatrix} s_1 \\ s_2 \end{bmatrix}$

SVD tells us which parameters are important, might be very few!

Later see that # parameters also scales much better
This form of wavefunction known as matrix product state (MPS)

Why? Amplitude a product of matrices:

$$|\Psi\rangle = \sum_{s_1, \alpha, \alpha', s_2} A_{s_1 \alpha} D_{\alpha \alpha'} B_{\alpha' s_2} |s_1\rangle |s_2\rangle$$

Schollwöck, Ann. of Phys. 326, 96 (2011)
Can use this form of the wavefunction to compute entanglement

\[ \psi_{s_1 s_2} = |s_1\rangle \otimes |s_2\rangle \]

\[ |\Psi\rangle = \sum_{s_1, \alpha, \alpha', s_2} A_{s_1 \alpha} D_{\alpha \alpha'} B_{\alpha' s_2} |s_1\rangle |s_2\rangle \]

Define \( \lambda_{\alpha} \) to be \( \lambda_{\alpha} = D_{\alpha \alpha} \) (\( \lambda_{\alpha} \) singular values or Schmidt weights)

Then entanglement defined to be

\[ S = -\sum_{\alpha} \lambda_{\alpha}^2 \log(\lambda_{\alpha}^2) \]
We’ll use the SVD to study the entanglement of a two-site wavefunction

cd tutorial/02_two_site

1. Read through **two.cc**; compile; and run

2. Run the program with different values for $\beta$

3. SVD the wavefunction $\psi$

   ```
   ITensor A(s1),D,B;
   auto spectrum = svd(psi,A,D,B);
   PrintData(D);
   ```

3. Compute the entanglement entropy using the density matrix spectrum returned by svd.

   $n^{th}$ eigenvalue (1-indexed): $\text{spectrum.eig}(n); //n=1,2,3,...$

   Number of eigenvalues: $\text{spectrum.size()}$
Matrix Product States in ITensor

\[ |\Psi \rangle = \begin{array}{c}
  s_1 \\
  \downarrow \\
  \downarrow \\
  s_2 \\
  \downarrow \\
  \downarrow \\
  s_3 \\
  \downarrow \\
  \downarrow \\
  s_4
\end{array} \]

Create an MPS

```
auto sites = SpinHalf(N);
auto psi = MPS(sites);
```

Use DMRG to optimize

```
dmrg(psi,H,sweeps);
```
Or, can initialize to product state

\[ |\Psi\rangle = \begin{array}{cccc}
    s_1 & s_2 & s_3 & s_4 \\
    \uparrow & \downarrow & \uparrow & \downarrow 
\end{array} \]

```cpp
auto sites = SpinHalf(N);
auto init = InitState(sites);
for(auto n : range1(N))
    {
    if(odd(n)) init.set(n,"Up");
    else init.set(n,"Dn");
}
auto psi = MPS(init);
```
Recall MPS can be put into various "gauges"

\[ |\Psi\rangle = \quad = \quad "orthogonality center" \text{ at site 2} \]

Where

\[ \quad = \quad \quad = \quad \quad = \quad \]
Use `.position(j)` method to efficiently change gauge

$$|\Psi\rangle = \begin{array}{c}
\begin{array}{ccc}
\text{Yellow} & \text{Red} & \text{Purple} \\
\text{Red} & \text{Purple} & \text{Purple}
\end{array}
\end{array}$$

```python
psi.position(2);
```
Use `.position(j)` method to efficiently change gauge

\[ |\Psi\rangle = \begin{array}{c}
\text{\textcolor{orange}{Orange}}
\end{array} \begin{array}{c}
\text{\textcolor{yellow}{Yellow}}
\end{array} \begin{array}{c}
\text{\textcolor{red}{Red}}
\end{array} \begin{array}{c}
\text{\textcolor{purple}{Purple}}
\end{array} \]

psi.position(3);
Use .position(j) method to efficiently change gauge

\[ |\Psi\rangle = \text{Diagram with yellow and red states connected} \]

psi.position(4);
Use `.position(j)` method to efficiently change gauge

\[ |\Psi\rangle = \quad \text{\includegraphics{diagram.png}} \]

psi.position(3);
Gauged form of MPS more efficient

$$|\Psi\rangle = \begin{array}{c}
\text{site 1} \\
\text{site 2} \\
\text{site 3} \\
\text{site 4}
\end{array}$$

Measurement of single-site operator

$$s'_{i_1} = O_{s'_{i_1}s_{i_1}}$$

$$\langle \Psi | O | \Psi \rangle = \begin{array}{c}
\text{site 1} \\
\text{site 2} \\
\text{site 3} \\
\text{site 4}
\end{array}$$

Only $O(1)$ tensor contractions needed [versus $O(N)$]
Gauged form of MPS more efficient

\[ |\Psi\rangle = \text{Diagram of MPS}\]

Measurement of single-site operator

\[ s'_1 = \mathcal{O}_{s_1', s_1}\]

\[ \langle \Psi | \mathcal{O} | \Psi \rangle = \text{Diagram of measurement}\]

Only \(O(1)\) tensor contractions needed [versus \(O(N)\)]
Gauged form of MPS more efficient

\[ |\Psi\rangle = \text{Diagram of MPS state} \]

Measurement of single-site operator

\[ s'_1 = O_{s'_1 s_1} \]

\[ \langle \Psi | O | \Psi \rangle = \text{Diagram of measurement} \]

Only $O(1)$ tensor contractions needed [versus $O(N)$]
Gauged form of MPS more efficient

\[ |\Psi\rangle = \begin{array}{c}
\text{Red}\vline\
\text{Red}\vline\
\text{Purple}\vline\
\text{Purple}\vline\
\end{array} \]

Measurement of single-site operator

\[ \frac{s'_1}{s_1} = \mathcal{O}_{s'_1 s_1} \]

\[ \langle \Psi | \mathcal{O} | \Psi \rangle = \begin{array}{c}
\text{Red}\vline\
\text{Red}\vline\
\end{array} \]

Only \( O(1) \) tensor contractions needed [versus \( O(N) \)]
How much have we gained?
How much have we gained?

Link index runs from 1...m
(from SVD)
How much have we gained?

Two site indices going from 1...d

Link index runs from 1...m (from SVD)
How much have we gained?

Two site indices going from 1...d

Link index runs from 1...m (from SVD)

Computational cost $\sim d^2 m$

Compare to $\sim (N m^3 d)$ for full MPS contraction
Measuring observable in ITensor

• Use `psi.A(j)` to obtain MPS tensor at site j

• Use `sites.op("Sz",j)` to obtain Sz at site j

• Use `prime(psi.A(j),Site)` to prime only the site/physical index of an MPS tensor

```cpp
auto sz_1 = ( dag(prime(psi.A(1),Site)) * sites.op("Sz",1) * psi.A(1) ).real();
```
Just as we can measure one-site operators, can measure two-site operators.

Recall:
Just as we can measure one-site operators, we can measure two-site operators.

Recall:
Since two “center” sites have orthogonal environment, ok to apply operators:
Since two “center” sites have orthogonal environment, ok to apply operators:
Since two “center” sites have orthogonal environment, ok to apply operators:

SVD & regroup
Would NOT be ok on another bond without regauging

SVD truncation not globally optimal except at orthogonality center
Would NOT be ok on another bond without regauging

SVD truncation not globally optimal except at orthogonality center
Q: What can we do with this capability?
Q: What can we do with this capability?

A: For short-ranged Hamiltonians, can time evolve
Trick is to use Trotter decomposition

Useful for Hamiltonians of the form

\[ H = H_1 + H_2 + H_3 + \ldots \]

For example

\[ H = \sum_j S_j \cdot S_{j+1} \]

\[ = (S_1 \cdot S_2) + (S_2 \cdot S_3) + (S_3 \cdot S_4) \]
For a small time step $\tau$

$$e^{-\tau H} \simeq e^{-\tau H_1/2} e^{-\tau H_2/2} e^{-\tau H_3/2} \ldots$$

$$\ldots e^{-\tau H_3/2} e^{-\tau H_2/2} e^{-\tau H_1/2} + O(\tau^3)$$
Diagramatically,
Diagrammatically,
Diagrammatically,
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Apply to MPS as follows:
Interesting applications: \[ |\psi'\rangle = e^{-\tau H} |\psi\rangle \]
Interesting applications:

\[ |\psi'\rangle = e^{-\tau H} |\psi\rangle \]

If \( \tau \) real (imaginary time evolution), enough steps will give ground state.
Interesting applications:

\[ |\psi'\rangle = e^{-\tau H} |\psi\rangle \]

If $\tau$ real (imaginary time evolution), enough steps will give ground state.

If $\tau$ imaginary, evolve in real time, study dynamics [1]

Interesting applications: \[ |\psi'\rangle = e^{-\tau H} |\psi\rangle \]

If \( \tau \) real (imaginary time evolution), enough steps will give ground state

If \( \tau \) imaginary, evolve in real time, study dynamics \[1\]

Evolving through imaginary time \( \beta/2 = 1/(2T) \) simulates finite temperature \[2\]

We’ll implement time evolution for the Heisenberg chain

cd tutorial/05_gates

1. Read through `gates.cc`; compile (make); and run

2. Apply the gate G to the MPS bond tensor AA using the ITensor contraction $\ast$ operator

3. Reset the prime level back to zero using AA’s `.noprime()` class method

4. Try increasing the total time “$t_{total}$” to imaginary time evolve toward the ground state.
   (Exact energy for 20 sites: $E_0 = -8.6824733317$)

5. Try improving the logic of the code to put $D$ into $U$ or $V$ depending on left/right sweep [and remove `.position(b)`]