ITensor
ITensor – Key features

- Tensor indices carry extra info and matching indices automatically contract
- Real & complex tensors, dense & sparse tensors all share same interface
- Useful for MPS/DMRG and 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 network calculations. Its efficiency and flexibility make it very useful for research-grade simulations.

Features include:

- Indices have unique ids: 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
- Interface only uses friendly, productive subset of the C++ language

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

\[ A \otimes C = A \otimes B \otimes C \otimes D \]
Wide Range of Applications
DMRG in 1D and 2D (cylinders)

3-band model of high-Tc (White, Scalapino)
Tensor renormalization group (TRG)

Partition function of 2D classical model

Compute for exponentially large systems, no Monte Carlo error bars!
Machine Learning with Tensor Networks

Recognizing handwritten digits with MPS

\[ f(x) = W \Phi(x) \]
High-Precision Chemistry

"Sliced basis" DMRG

Linear scaling method, able to treat strong correlation
Users express algorithms,
ITensor handles computation
ITensor – Philosophy

Strive for ideal of translating diagrams directly into code

Diagram indices unordered, so ITensor indices also unordered
Einstein summation approach
(not ITensor code)

\[ C[c,d,e] = A[a,b,c] \times B[b,d,a,e] \]
A = ITensor(a, b, c);
B = ITensor(a, b, d, e);
C = A * B;

\[ a, b, c, \ldots \text{ are Index objects} \]
Making indices

\[
\begin{align*}
a &= \text{Index}("a", 4); \\
b &= \text{Index}("b", 3); \\
c &= \text{Index}("c", 8); \\
A &= \text{ITensor}(a, b, c);
\end{align*}
\]
Setting tensor elements

\[
\begin{align*}
A &= \text{ITensor}(a,b,c); \\
A\text{.set}(b(3),a(2),c(4), 8.723); \\
A\text{.set}(b(1),a(2),c(1),-2.212); \\
A\text{.set}(a(3),c(3),b(2), 1.007); \\
\end{align*}
\]
Setting tensor elements

Has the same effect

\[ \begin{align*}
\text{c} &= 4 \\
\text{a} &= 2 \\
\text{b} &= 3 \\
\end{align*} \]

= 8.723

\[ \text{A} = \text{ITensor}(a, b, c); \]

A.set(b(3), a(2), c(4), 8.723);
A.set(a(2), b(3), c(4), 8.723);

Has the same effect
The * operator contracts all matching indices

A = ITensor(a,b,c);
B = ITensor(a,b,d,e);
C = A * B;
The \( \ast \) operator contracts all matching indices

\[
A = \text{ITensor}(a,b,c); \\
B = \text{ITensor}(a,b,d,e); \\
C = B \ast A;
\]

Commutative but not associative
Adding "just works"

\[ E = \text{ITensor}(a,b,c); \]
\[ F = \text{ITensor}(b,a,c) \]
\[ R = F + E; \]
The * operator is "hungry" to contract

Use primes to prevent indices from contracting
Use *primes* to prevent indices from contracting

```cpp
auto W = ITensor(i,s,j);
```
Use *primes* to prevent indices from contracting

```cpp
auto W = ITensor(i, s, j);
```

How to compute?
Use *primes* to prevent indices from contracting

```cpp
auto W = ITensor(i, s, j);
```

How to compute?

Prime one copy of *s*
Use *primes* to prevent indices from contracting

\[ W = \text{ITensor}(i,s,j); \]

\[ R = W \ast \text{prime}(W,s); \]
ITensors have a dynamic storage type hidden behind an opaque storage pointer

Operations on ITensors passed to storage, unmasking the type
ITensors have a dynamic storage type hidden behind an opaque storage pointer

Operations on ITensors passed to storage, unmasking the type
ITensors have a dynamic storage type hidden behind an opaque storage pointer.

```
ITensor
```

```
multiply(.DenseReal, 5.0)
```

Operations on ITensors passed to storage, unmasking the type.
ITensors have a dynamic storage type hidden behind an opaque storage pointer

\[
\text{ITensor} \rightarrow \text{multiply( DenseReal , 5.0 )}
\]

Operations on ITensors passed to storage, unmasking the type

"Plug in" new storage types and behaviors
- sparse storage, distributed storage, ...
- test new algorithms
- lazy storage types
DMRG With ITensor
```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");
```
int N = 100;
auto sites = SpinOne(N);
int N = 100;
auto sites = SpinOne(N);
```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);
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 \texttt{dmrg.cc} sample code (100 site S=1 Heisenberg chain)
   - \texttt{cd sample/} inside \texttt{ITensor} folder
   - \texttt{make dmrg}
   - \texttt{./dmrg}

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

3. Compute \( E = \frac{(E_{110} - E_{100})}{10} \) [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 \( e_0 \cong -1.401484038971(4) \). Open-ended \( S = 1 \) chains have effective
Extending DMRG
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;
    }
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);

\[
H = \sum_j \left( \frac{1}{2} S_j^+ S_{j+1}^- + \frac{1}{2} S_j^- S_{j+1}^+ + J_z S_j^z S_{j+1}^z \right)
\]
Modify Terms in Hamiltonian

```
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);
Operators on more than 2 sites

\[ H = \sum_j \left( S_j^+ S_{j+1}^- + S_j^- S_{j+1}^+ \right) + \sum_j S_j^z S_{j+1}^z S_{j+2}^z S_{j+3}^z \]
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:

\[ \psi_s \rightarrow \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} \]

**Using ITensor:**

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

First choose $|\psi\rangle = |\uparrow\rangle$  

\[ \begin{align*} 1 = 1 \end{align*} \]

```cpp
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 // ITensor constructor

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:

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

Set their components:

```plaintext
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 \( \hat{S}_x |\psi\rangle = |\phi\rangle \)

\[
(\hat{S}_x)_{s'}^s \psi_s =
\]

In code,

```cpp
ITensor phi = Sx * psi;
```

The \( \ast \) operator *contracts all matching indices*

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

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

\[
(\hat{S}_x)_{s'}^s \psi_s = \begin{array}{c}
\text{state} \\
\text{state}
\end{array}
\]

ITensor $\phi = Sx \ast \psi$;

PrintData($\phi$);

\[
\phi = \begin{array}{c}
\text{ITensor } r=1: (s,2,\text{Link,273})' \\
(2) 0.500
\end{array}
\]
More interesting $\psi_s$: choose $\theta = \pi/4$ and

\[
1 = \cos \theta / 2
\]

\[
2 = \sin \theta / 2
\]

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

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

Real theta = Pi/4.;

```plaintext
psi = ITensor r=1: (s,2,Link,273)
psi.set(s(1),cos(theta/2))
psi.set(s(2),sin(theta/2))
PrintData(psi);
```

(1) 0.92388
(2) 0.38268
Diagrammatically, measurements (expectation values) look like:

\[ \langle \psi | \hat{S}_z | \psi \rangle \]

For convenience, make:

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

Calculate expectation values:

```cpp
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 $|\phi\rangle = \hat{S}_x |\psi\rangle$

```cpp
auto phi = Sx * psi;
phi.noprime();
auto olap = (dag(psi)*phi).real();
```

4. Try normalizing phi before computing overlap

```cpp
phi /= norm(phi);
```
Most general two-site wavefunction is

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

Amplitudes are a rank-2 tensor
Let’s make a singlet

\[
\begin{array}{c}
1 \quad 2 \\
\end{array}
\begin{array}{c}
\\quad \\
1 \quad 2 \\
\\quad \\
\end{array}
= \frac{1}{\sqrt{2}}
\]

\[
\begin{array}{c}
2 \quad 1 \\
\end{array}
\begin{array}{c}
\\quad \\
2 \quad 1 \\
\\quad \\
\end{array}
= -\frac{1}{\sqrt{2}}
\]

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 $H$

$$
\hat{H} = \begin{pmatrix}
  \begin{array}{c}
  \text{Blue} \\
  \text{Red} \\
  \text{Green}
  \end{array}
\end{pmatrix} + \frac{1}{2} \begin{pmatrix}
  \begin{array}{c}
  \text{Blue} \\
  \text{Red} \\
  \text{Green}
  \end{array}
\end{pmatrix} + \frac{1}{2} \begin{pmatrix}
  \begin{array}{c}
  \text{Blue} \\
  \text{Red} \\
  \text{Green}
  \end{array}
\end{pmatrix}
$$

Showing Index labels

$$
\hat{H} = \begin{pmatrix}
  \begin{array}{c}
  s'_1 \\
  s'_2 \\
  s_1 \\
  s_2
  \end{array}
\end{pmatrix}
$$

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

\[ \hat{H} |\psi\rangle = \begin{bmatrix} s_1' & s_2' \\ s_1 & s_2 \end{bmatrix} = \begin{bmatrix} \hat{H} \\ \psi \end{bmatrix} \]

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

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

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

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

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

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

Real \( E = (\text{dag}(\text{prime}(\psi)) \times H \times \psi).\text{real}(); \)

Print(E);
//prints: \( E = -0.75 \)
For an arbitrary Hamiltonian, can find ground state by doing imaginary time evolution

\[ e^{-\beta H} \left| \Psi_{\text{init}} \right\rangle \propto \left| \Psi_0 \right\rangle \]
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();
```
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

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

$$
\begin{bmatrix}
1/2 & -1/2 & 1/2 \\
1/2 & -1/2 & -1/2 \\
1/2 & 1/2 & 1/2 \\
1/2 & 1/2 & -1/2 \\
\end{bmatrix}
\begin{bmatrix}
0.933 & 0 & 0 \\
0 & 0.300 & 0 \\
0 & 0 & 0.200 \\
\end{bmatrix}
\begin{bmatrix}
0.707107 & 0.707107 & 0 \\
-0.707107 & 0.707107 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
$$

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

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

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

$= M_2 =$

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

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

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

\[
M_2 = \begin{bmatrix}
0.707107 & 0.707107 & 0 \\
-0.707107 & 0.707107 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\| M_2 - M \|_2^2 = 0.04 = (0.2)^2
\]
Keep fewer and fewer elements of $D$:

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

\[= M_3 = \begin{bmatrix}
0.329773 & 0.329773 & 0 \\
0.329773 & 0.329773 & 0 \\
0.329773 & 0.329773 & 0 \\
0.329773 & 0.329773 & 0
\end{bmatrix}\]

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

\[
\begin{bmatrix}
1/2 & -1/2 & 1/2 \\
1/2 & -1/2 & -1/2 \\
1/2 & 1/2 & 1/2 \\
1/2 & 1/2 & -1/2
\end{bmatrix}
\begin{bmatrix}
0.933 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
0.707107 & 0.707107 & 0 \\
-0.707107 & 0.707107 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.329773 & 0.329773 & 0 \\
0.329773 & 0.329773 & 0 \\
0.329773 & 0.329773 & 0 \\
0.329773 & 0.329773 & 0
\end{bmatrix}
\]

\[
\|\|\mathbf{M}_3 - \mathbf{M}\|\|^2 = 0.13 = (0.3)^2 + (0.2)^2
\]
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

\[ \psi_{s_1 s_2} = \begin{array}{c}
\end{array} \]

Can treat as a matrix:

\[ \psi_{s_1 s_2} = \begin{array}{c}
\end{array} \]
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

auto $A = \text{ITensor}(s1)$
ITensor $D,B$;

Call SVD function

$\text{svd}(\psi,A,D,B)$;
What have we gained from SVD?

Generic two-spin wavefunction (say spin S):

$$(2S+1)^2$$ parameters

Not clear which parameters important, unimportant

Compressed wavefunction:

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)
Different equivalent matrix product state forms, or "gauges"

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

\[ |\Psi\rangle = \sum_{s_1 \alpha s_2} \psi_{s_1 \alpha} B_{\alpha s_2} |s_1\rangle |s_2\rangle \]
Different equivalent matrix product state forms, or "gauges"

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

\[ |\Psi\rangle = \sum_{s_1 \alpha s_2} A_{s_1 \alpha} \psi_{\alpha s_2} |s_1\rangle |s_2\rangle \]
Matrix Product States in ITensor
Matrix Product States in ITensor

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

Create an MPS

\begin{verbatim}
auto sites = SpinHalf(N);
auto psi = MPS(sites);
\end{verbatim}

Use DMRG to optimize

\begin{verbatim}
dmrg(psi,H,sweeps);
\end{verbatim}
Or, can initialize to product state

\[ |\Psi\rangle = \begin{array}{cccc}
\uparrow & \downarrow & \uparrow & \downarrow \\
S_1 & S_2 & S_3 & S_4 \\
\end{array} \]

```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"

\[ |\Psi\rangle = \begin{array}{c}
\begin{array}{cccc}
\text{white} & \text{white} & \text{white} & \text{white}
\end{array}
\end{array} = \begin{array}{c}
\begin{array}{cccc}
\text{yellow} & \text{red} & \text{purple} & \text{purple}
\end{array}
\end{array} \]

"orthogonality center" at site 2

Where

\[ \begin{array}{c}
\begin{array}{c}
\text{yellow}
\end{array}
\end{array} = \begin{array}{c}
\begin{array}{c}
\text{yellow}
\end{array}
\end{array} \quad \begin{array}{c}
\begin{array}{c}
\text{purple}
\end{array}
\end{array} = \begin{array}{c}
\begin{array}{c}
\text{purple}
\end{array}
\end{array} \]
Use `.position(j)` method to efficiently change gauge

\[ |\Psi\rangle = \begin{array}{c}
\text{\textcolor{orange}{orange}} \quad \text{\textcolor{red}{red}} \quad \text{\textcolor{purple}{purple}} \quad \text{\textcolor{purple}{purple}}
\end{array} \]

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

\[ |\Psi\rangle = \begin{array}{c}
| & | & | & | \\
\text{yellow} & \text{yellow} & \text{red} & \text{purple} \\
\end{array} \]

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

\[ |\Psi\rangle = \begin{array}{ccccc}
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } \\
\end{array} \]

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

$$|\Psi\rangle = \begin{array}{cccc}
\text{Orange} & \text{Orange} & \text{Red} & \text{Purple} \\
\end{array}$$

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

\[ |\Psi\rangle = \begin{array}{c}
\text{\textcolor{red}{red}}
\end{array} \begin{array}{c}
\text{\textcolor{purple}{purple}}
\end{array} \begin{array}{c}
\text{\textcolor{purple}{purple}}
\end{array} \begin{array}{c}
\text{\textcolor{purple}{purple}}
\end{array} \]

Measurement of single-site operator

\[
\begin{array}{c}
\text{\textcolor{purple}{purple}}
\end{array}
\begin{array}{c}
\text{\textcolor{gray}{gray}}
\end{array}
\begin{array}{c}
\text{\textcolor{red}{red}}
\end{array}
\]

\[ = \mathcal{O}_{s_1} \]

\[ = \mathcal{O}_{s_1' s_1} \]

\[ \langle \Psi | \mathcal{O} | \Psi \rangle = \begin{array}{c}
\text{\textcolor{red}{red}}
\end{array} \begin{array}{c}
\text{\textcolor{purple}{purple}}
\end{array} \begin{array}{c}
\text{\textcolor{purple}{purple}}
\end{array} \begin{array}{c}
\text{\textcolor{purple}{purple}}
\end{array} \begin{array}{c}
\text{\textcolor{gray}{gray}}
\end{array} \]

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

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

Measurement of single-site operator

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

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

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

\[ |\Psi\rangle = \odot \]

Measurement of single-site operator

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

\[ \langle \Psi | \mathcal{O} | \Psi \rangle = \odot \]

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

\[ |\Psi\rangle = \begin{array}{l}
\hline
\hline
\hline
\hline
\end{array} \]

Measurement of single-site operator

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

\[ \langle \Psi | O | \Psi \rangle = \begin{array}{l}
\hline
\hline
\hline
\hline
\end{array} \]

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

```
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:

\[
\begin{align*}
\text{[Diagram: two-site operators]} &= \text{[Diagram: one-site operator]} \\
\text{[Diagram: two-site operators]} &= \text{[Diagram: one-site operator]}
\end{align*}
\]
Just as we can measure one-site operators, we can measure two-site operators.

Recall:

\[
\begin{align*}
\text{\includegraphics[width=0.2\textwidth]{image1}} &= \text{\includegraphics[width=0.2\textwidth]{image2}} \\
\text{\includegraphics[width=0.2\textwidth]{image3}} &= \text{\includegraphics[width=0.2\textwidth]{image4}}
\end{align*}
\]
We’ll implement local measurements for the Heisenberg chain sample/dmrg.cc

1. Read through **dmrg.cc** again; compile (make); and run

2. Write a loop that measures \(<Sz>\) on each site.
   First, create a loop from 1 to N:
   
   ```
   for(int i : range1(N)) {
   ... }
   ```

3. Inside the loop, call: `psi.position(i);` to **gauge** the MPS to position, or site i

4. Obtain the \(i^{th}\) MPS tensor by calling: `psi.A(i)`
   and obtain the \(S^z_i\) operator by calling: `sites.op("Sz",i)`

5. Measure \(<Sz>\) by contracting the following diagram: