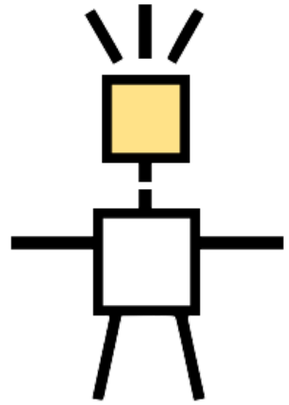
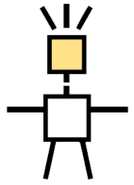
ITensor



E.M. Stoudenmire

Apr 2018 - Sao Paulo

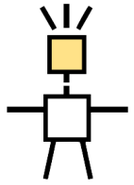
FLATIRON INSTITUTE
SIMONS FOUNDATION



ITensor – Key features

SIMONS FOUNDATION

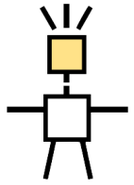
- Tensor indices carry extra info and matching indices automatically contract



ITensor – Key features

SIMONS FOUNDATION

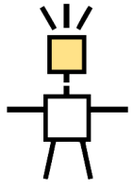
- Tensor indices carry extra info and matching indices automatically contract
- Real/complex and dense/sparse/block-sparse tensors



ITensor – Key features

SIMONS FOUNDATION

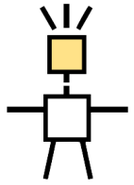
- Tensor indices carry extra info and matching indices automatically contract
- Real/complex and dense/sparse/block-sparse tensors
- Useful not only for MPS/DMRG but general tensor network algorithms



ITensor – Key features

SIMONS FOUNDATION

- 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



ITensor – Key features

SIMONS FOUNDATION

- 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 Hamiltonians from code resembling hand written mathematical notation

Website http://itensor.org

Home
News
Learn
Discuss
About ITensor

ITensor

Introduction

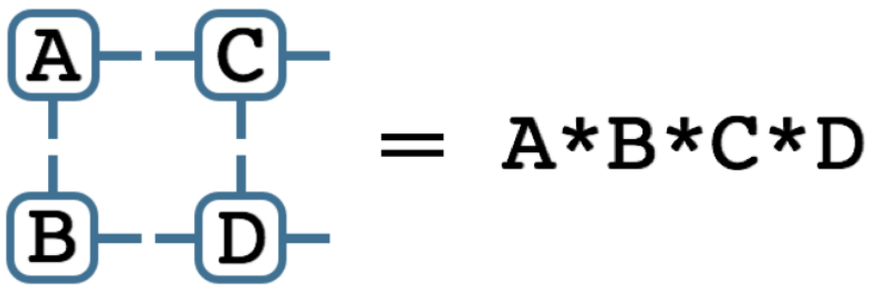
ITensor—Intelligent Tensor—is a C++ library for implementing tensor product wavefunction calculations. It is efficient and flexible enough to be used for research-grade simulations.

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



Latest version is v2.0.9
Clone from github (preferred)
Download: tar.gz, zip
Report bugs: code
website
Follow: @ITensorLib

Recent News

- ITensor at Simons Many-Electron Summer School 2016
- Tutorial on Fermions and Jordan-Wigner Mapping
- New Discussion Forum
- Version 2.0 released!
- ITensor at 2016 Sherbrooke Summer School

Basics of C++

"Bare bones" ITensor program

program.cc

```
#include "itensor/all.h"

using namespace itensor;

int main()
    {

    <your code goes here>

    }
```

compiling and running

```
$ make

$ ./program
```

C++ is a *strongly typed* language

```
#include "itensor/all.h"

using namespace itensor;

int main()
    {

    int i = 5;
    print("i = ",i);

    string s = "a string";
    print("s = ",s);

    }
```

C++ is a *strongly typed* language

```
#include "itensor/all.h"

using namespace itensor;

int main()
    {

    int i = 5;
    print("i = ",i);

    string s = "a string";
    print("s = ",s);

    }
```

C++ is a *strongly typed* language

```
#include "itensor/all.h"

using namespace itensor;

int main()
    {

    int i = 5;
    print("i = ",i);

    string s = "a string";
    print("s = ",s);

    }
```

(Remaining examples will assume we're inside main function)

```
int i = 5;
print("i = ",i);

string s = "a string";
print("s = ",s);
```

C++ allows defining custom types

For example, ITensor defines a type called `Index`

Two ways to initialize a custom type

```
Index i1("index i1",5);          //old style

auto i2 = Index("index i2",8);   //new style
```

i1 and i2 will be variables of type `Index`

Looping over integers in C++

```
for(int n = 1; n <= 4; n += 1)
    {
    println("n = ",n);
    }
```

Will print:

```
n = 1
n = 2
n = 3
n = 4
```

ITensor library also provides helpful "range" and "range1" functions

```
for(auto n : range(4))
    {
    println("n = ",n);
    }
```

Will print:

```
n = 0
n = 1
n = 2
n = 3
```

ITensor library also provides helpful "range" and "range1" functions

```
for(auto n : range1(4))
    {
    println("n = ",n);
    }
```

Will print:

```
n = 1
n = 2
n = 3
n = 4
```

ITensor Tutorial

Consider a single-site wavefunction,
for example a spin 1/2

Single-site basis:

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

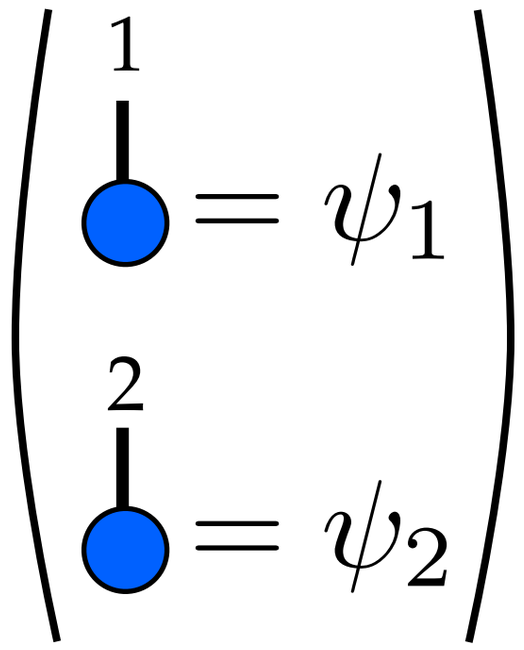
$$|s = 2\rangle = |\downarrow\rangle$$

Most general wavefunction for a spin 1/2:

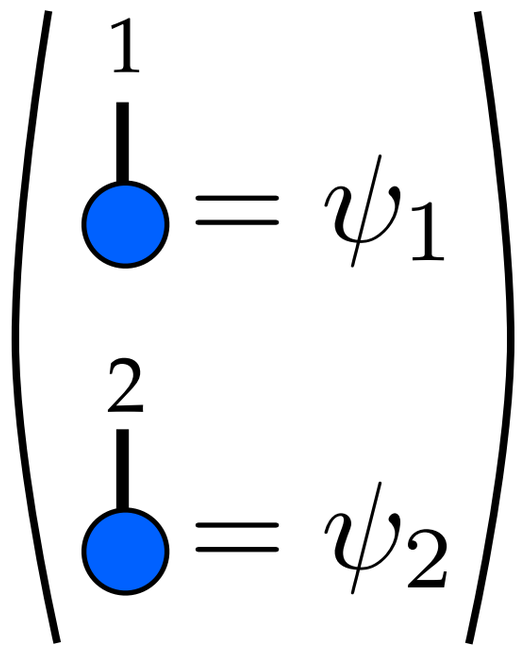
$$|\psi\rangle = \sum_{s=1}^{2} \psi_s |s\rangle$$

The ψ_s are complex numbers

Can view ψ_s as a tensor (one index)



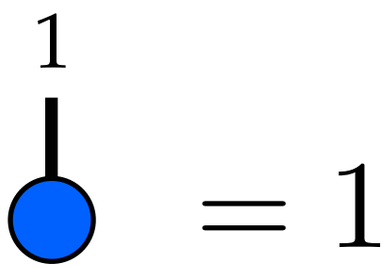
Single-site wavefunction as a tensor:

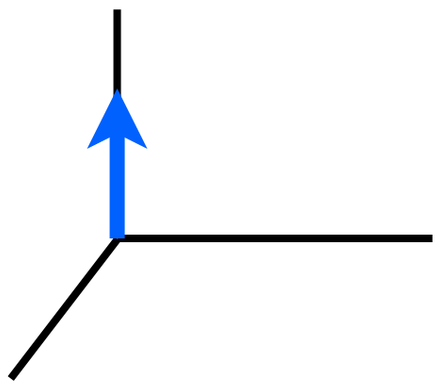


Using ITensor:

```
auto s = Index("index s",2);

auto psi = ITensor(s);
```

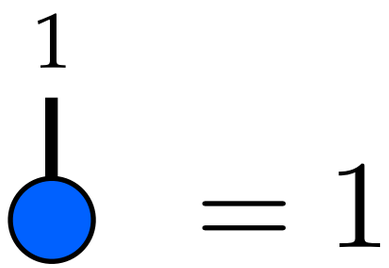
Now initialize ψ_s First choose $|\psi\rangle = |\uparrow\rangle$



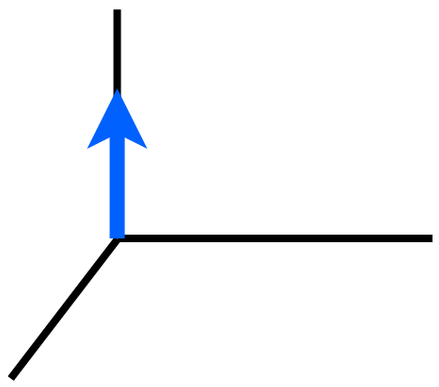


```
auto s = Index("s",2);
auto psi = ITensor(s);

psi.set(s(1), 1.0);

PrintData(psi);
```

Now initialize ψ_s First choose $|\psi\rangle = |\uparrow\rangle$



$= 1$

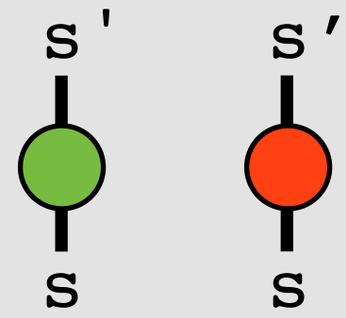


```
auto s = Index("s",2);
auto psi = ITensor(s);

psi.set(s(1), 1.0);

PrintData(psi);
```

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

Make some operators:

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

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



New ITensors start out set to zero

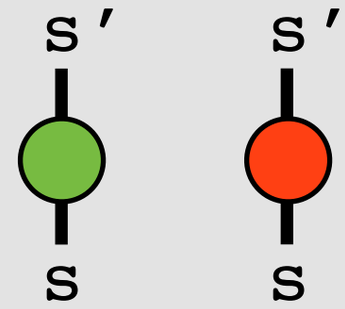
What does “prime” do?

`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:

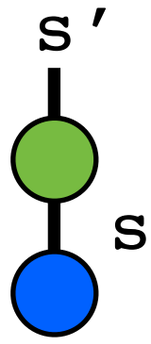
```
auto Sz = ITensor(s,prime(s));

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

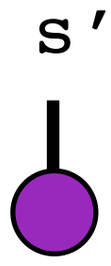


Set their components:

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

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

$$=$$ 

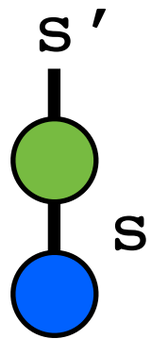
In code,

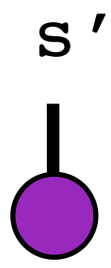
```
ITensor phi = Sx * psi;
```

The * operator ***contracts all matching indices***

Indices s and s' don't match because of their different prime levels

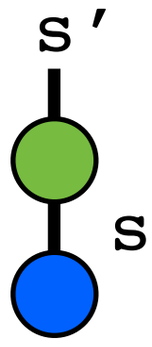
What state is phi ?

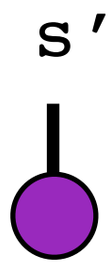
$$(\hat{S}_x)_{s'}{}^{s}\ \psi_s =$$ 

$=$ 


```
ITensor phi = Sx * psi;

PrintData(phi);
```

What state is phi ?

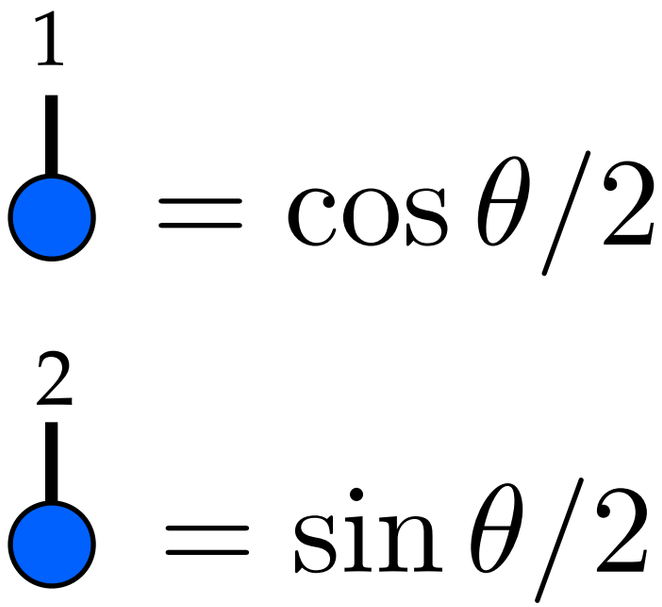
$$(\hat{S}_x)_{s'}{}^{s}\ \psi_s =$$ 

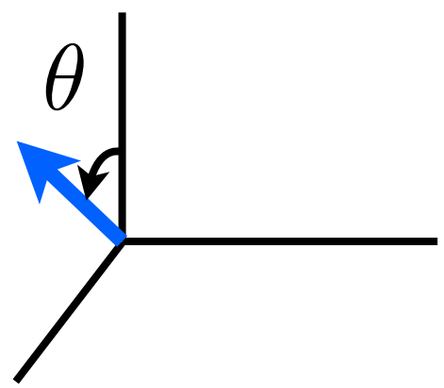
$=$ 


```
ITensor phi = Sx * psi;

PrintData(phi);
```

```
phi =
ITensor r=1: (s,2,Link,273)'
(2) 0.500
```

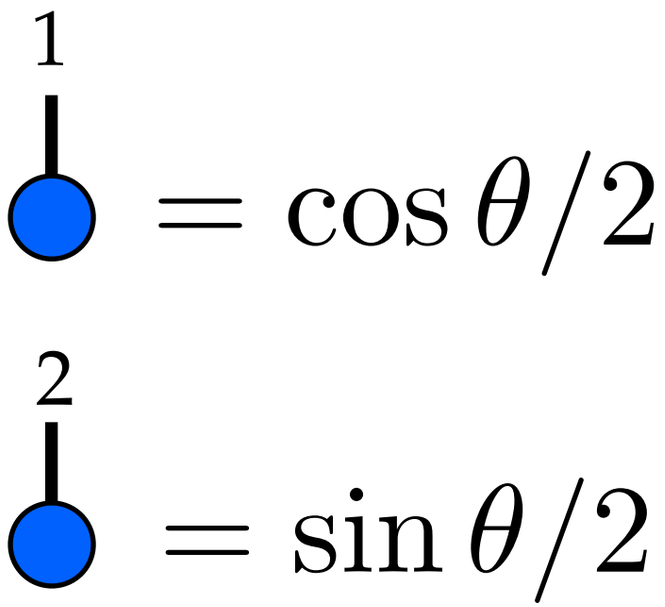
More interesting ψ_s : choose $\theta = \pi/4$ and

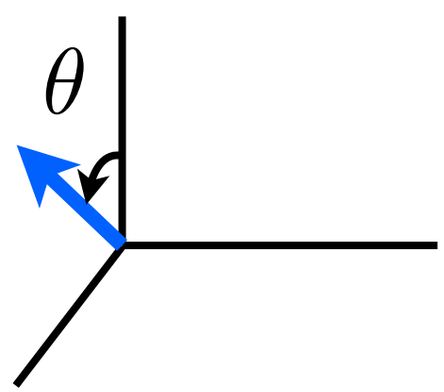





```
Real theta = Pi/4.;

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

PrintData(psi);
```

More interesting ψ_s : choose $\theta = \pi/4$ and






```
Real theta = Pi/4.;

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

PrintData(psi);
```

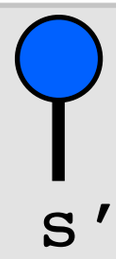
```
psi =
ITensor r=1: (s,2,Link,273)
(1) 0.92388
(2) 0.38268
```

Diagrammatically, measurements (expectation values) look like:

$$\langle\psi|\hat{S}_z|\psi\rangle$$

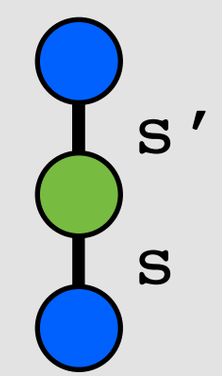
For convenience, make:

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



Calculate expectation values:

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

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



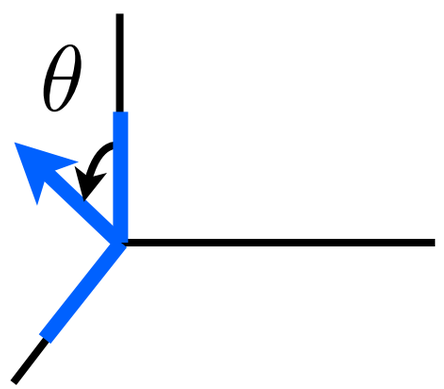
```
auto zz = (cpsi * Sz * psi).real();

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

```
auto a = Index("index a",4);
```

- Construct ITensor (indices a, b, c)

```
auto T = ITensor(a,b,c);
```

- Set ITensor components

```
T.set(a(2),c(3),b(1), 7.89);
```

- Prime an Index b ⟶ b′

```
prime(b)
```

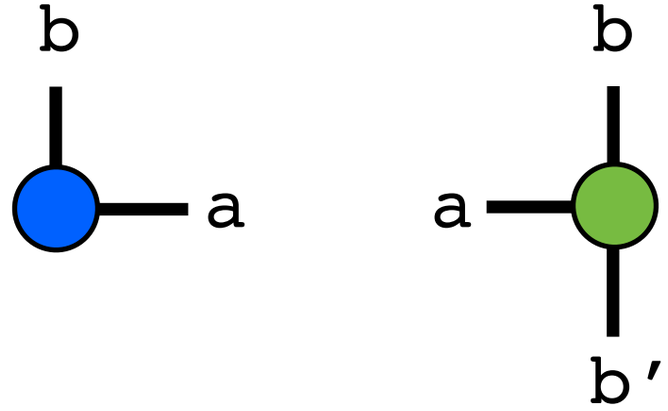
- The `*` operator automatically contracts matching Index pairs

Quiz:

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

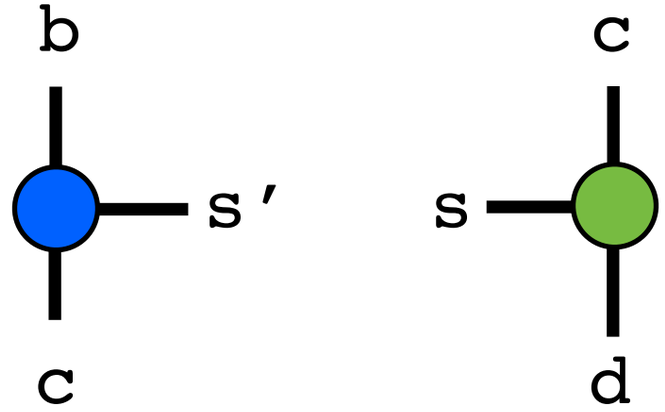
Quiz:

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 "**one.cc**", 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

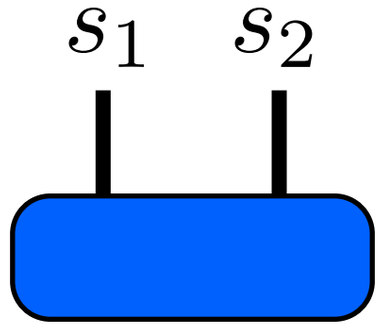
```
phi /= norm(phi);
```

Most general two-site wavefunction is

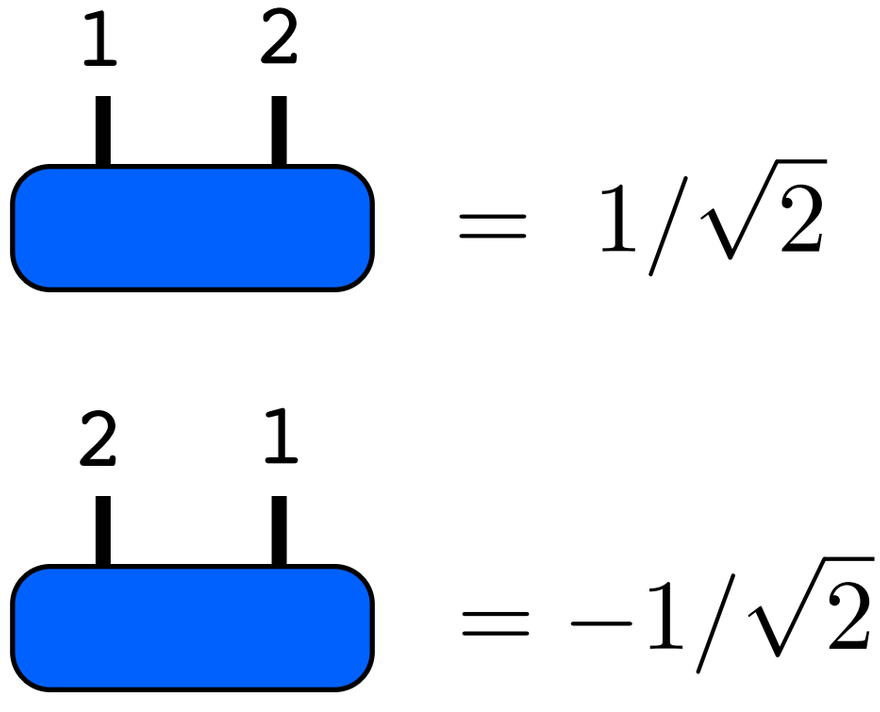
$$|\Psi\rangle = \sum_{s_1,s_2=1}^{2} \psi_{s_1 s_2} |s_1\rangle |s_2\rangle$$

Amplitudes are a rank-2 tensor

$$\psi_{s_1 s_2} =$$



Let's make a singlet



Using ITensor:

```
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:

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

Same result as:

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

Let's make the Heisenberg Hamiltonian $\hat{H} = \mathbf{S}_1 \cdot \mathbf{S}_2$

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

First create operators, for example S^+

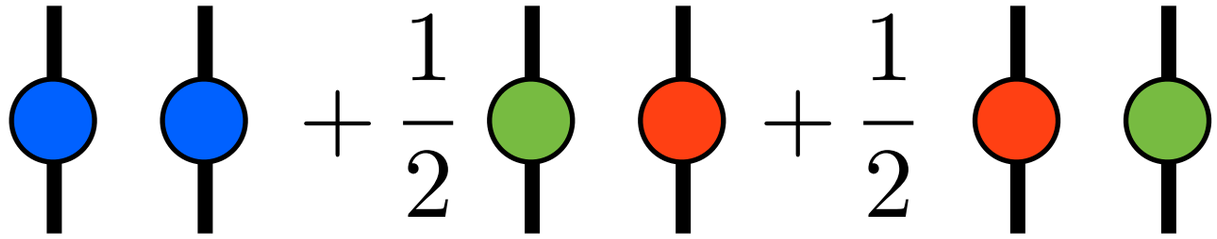
```
auto Sp1 = ITensor(s1,prime(s1));
Sp1.set(s1(2),prime(s1)(1), 1);
```

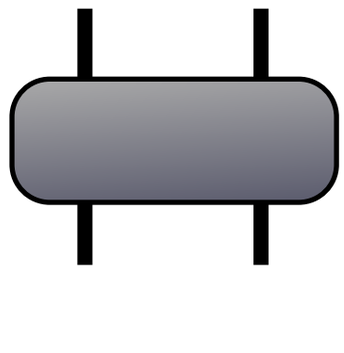
Multiply and add operators to make H:

```
auto H = Sz1*Sz2 + 0.5*Sp1*Sm2 + 0.5*Sm1*Sp2;
```

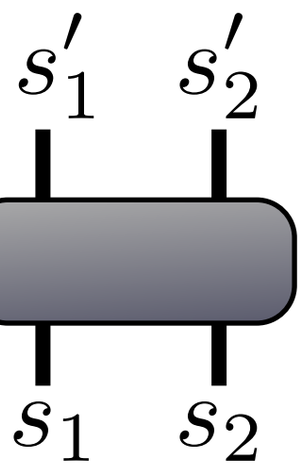
Tensor form of H

```
auto H = Sz1*Sz2 + 0.5*Sp1*Sm2 + 0.5*Sm1*Sp2;
```

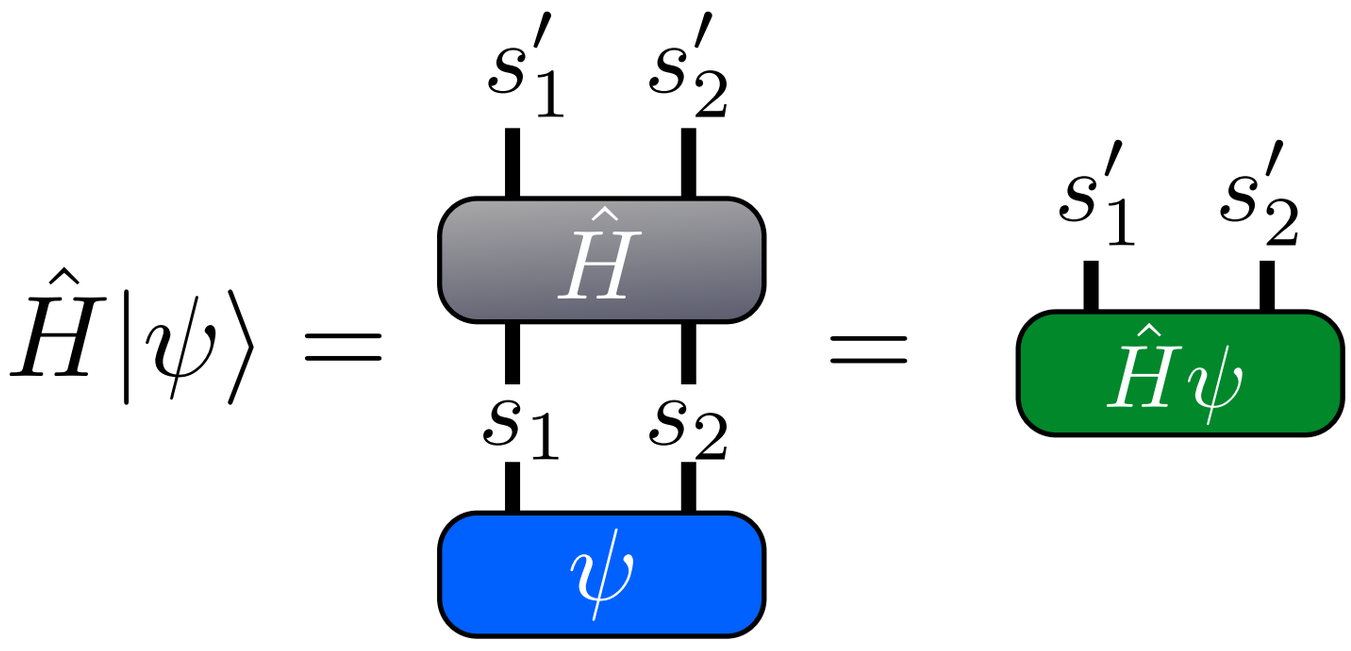
$$\hat{H} =$$  $$+ \frac{1}{2}$$ $$+ \frac{1}{2}$$

$$=$$ 

Showing Index labels

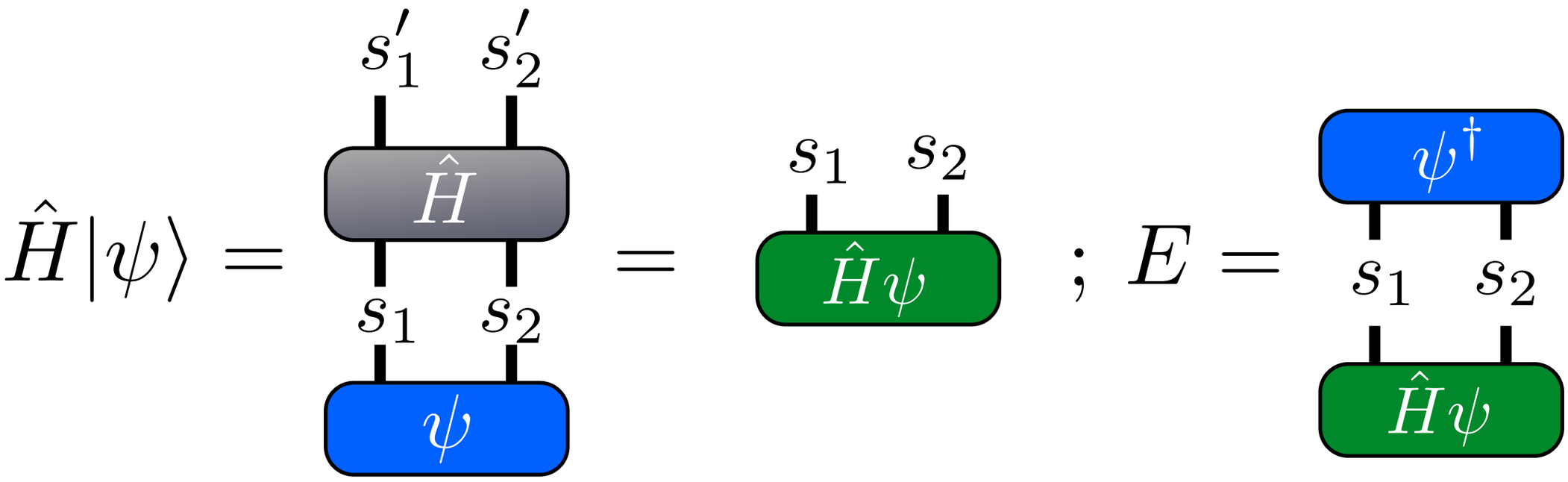
$$\hat{H} =$$ 

Compute singlet energy with this Hamiltonian:




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

Real E = (dag(Hpsi) * psi).real();
Print(E);
//prints: E = -0.75
```

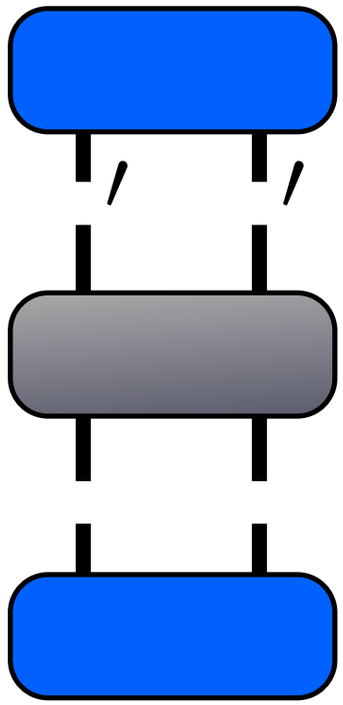
Compute singlet energy with this Hamiltonian:




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

Real E = (dag(Hpsi) * psi).real();
Print(E);
//prints: E = -0.75
```

Or compute energy in one shot:

$$E_{\text{sing}} =$$

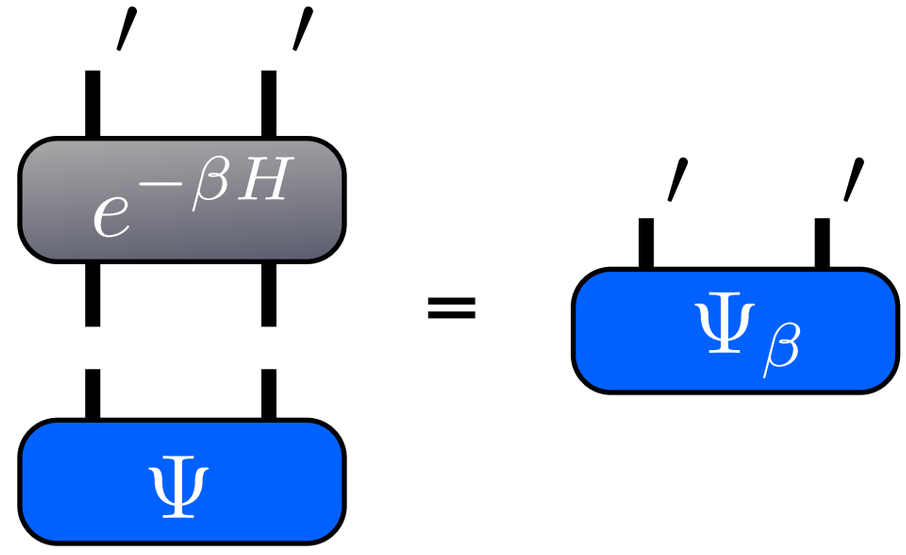


dag(prime(psi))

```
Real E = (dag(prime(psi)) * H * psi).real();

Print(E);
//prints: E = -0.75
```

For an arbitrary Hamiltonian, can find ground state by doing imaginary time evolution

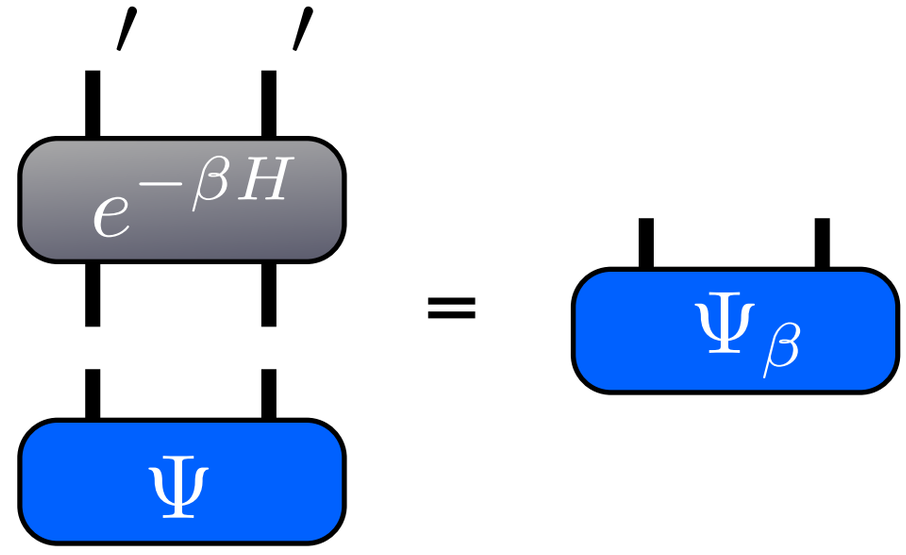
$$e^{-\beta H}|\Psi_{\text{init}}\rangle \propto |\Psi_0\rangle$$




```
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

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




```
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).

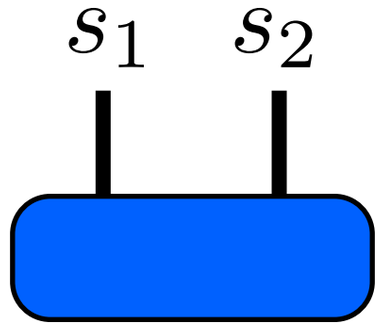
Matrix product states arise from compressing a one-dimensional wavefunction using the singular-value decomposition (SVD).

Let’s see how this works...

Recall:

Most general two-spin wavefunction

$$\psi_{s_1 s_2} =$$



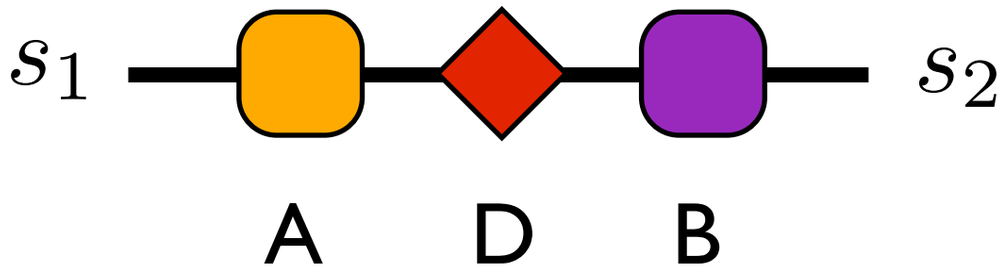
Can treat as a matrix:

$$\psi_{s_1 s_2} =$$

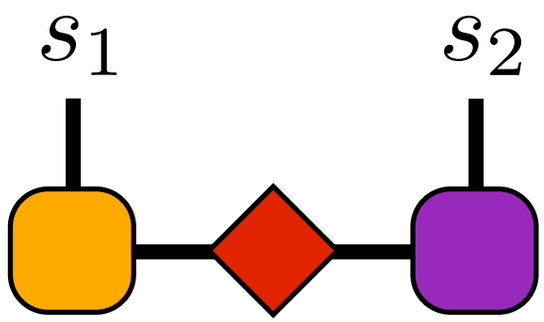
SVD this matrix:

$$\psi_{s_1 s_2} =$$

$=$

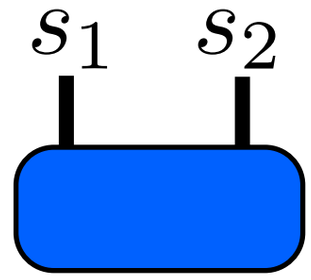


Bend lines back to look like wavefunction:



Using ITensor:

Say we have a two-site wavefunction psi

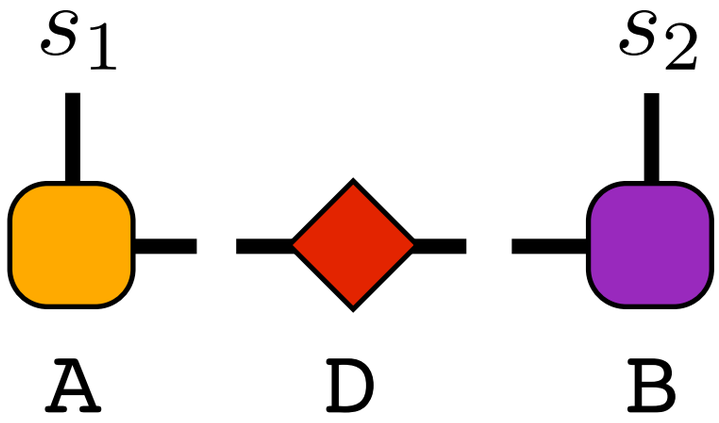


Declare A,D,B to hold results of SVD

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

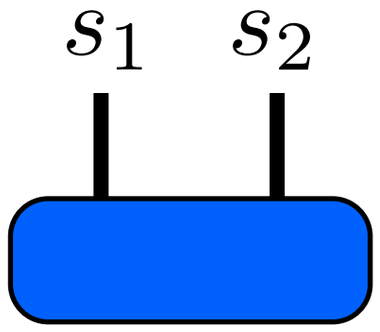
Call SVD function

```
svd(psi,A,D,B);
```



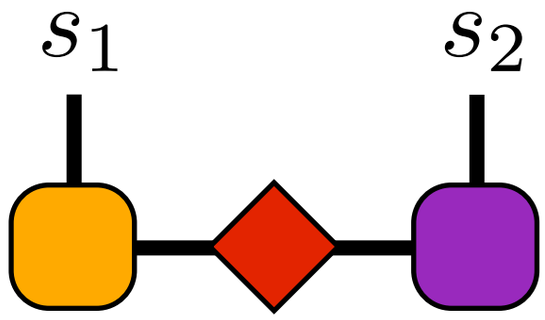
What have we gained from SVD?

Generic two-spin wavefunction (say spin S):



$(2S+1)^2$ parameters
Not clear which parameters important, unimportant

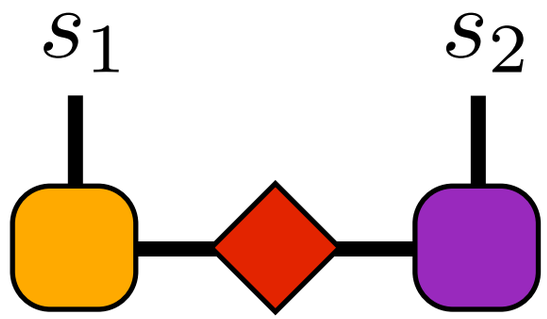
Compressed wavefunction:



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)

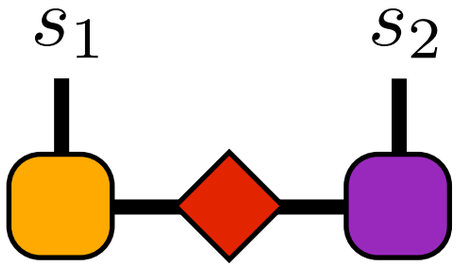


Why? Amplitude a product of matrices:

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

Schollwöck, Ann. of Phys. **326**, 96 (2011)

Can use this form of the wavefunction to compute entanglement

$\psi_{s_1 s_2} =$ 

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

Define λ_α to be $\lambda_\alpha = D_{\alpha\alpha}$ (λ_α 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 β

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):
   number of eigenvalues:

```
spectrum.eig(n); //n=1,2,3,...
spectrum.size();
```