ITensorMPOCompression

ITensorMPOCompression is a Julia language module based in the ITensors library. In general compression of Hamiltonian MPOs must be treated differntly than standard MPS compression. The root of the problem can be traced to the combination of both extensive and intensive degrees of freedom in Hamiltonian operators. If conventional compression methods are applied of Hamiltonain MPOs, one finds that two of the singular values will diverge as the lattice size increases, resulting in severe numerical instabilities. The recent paper

> *Local Matrix Product Operators:Canonical Form, Compression, and Control Theory* Daniel E. Parker, Xiangyu Cao, and Michael P. Zaletel **Phys. Rev. B** 102, 035147

contains a number of important insights for the handling of finite and infinite lattice MPOs. They show that the intensive degrees of freedom can be isolated from the extensive ones. If one only compresses the intensive portions of the Hamiltonian then the divergent singular values are removed from the problem. This module attempts to implement the algorithms described in the *Parker et. al.* paper. The initial release will support orthogonalization (canonical form) and truncation (SVD compression) of the finite lattice MPOs. A future releast will add support for handling iMPOs, or infinate lattice MPOs with a repeating unit cell. The techincal details are presented in the pdf document provided with this module: TechnicalDetails.pdf. A brief summary of the key functions of this module follows.

Block respecting QX decomposition

Finite MPO

A finite lattice MPO with *N* sites can expressed as

$$\hat{H} = \hat{W}^1 \hat{W}^2 \hat{W}^3 \cdots \hat{W}^{N-1} \hat{W}^N$$

where each
$\hat{W}^n$ is an operator-valued matrix on site *n*

Regular Forms

MPOs must be in the so called regular form in order for orthogonalization and compression to succeed. These forms are defined as follows:

$$\hat{W}_{upper} = \begin{bmatrix} \hat{\mathbb{I}} & \hat{\boldsymbol{c}} & \hat{d} \\ 0 & \hat{\boldsymbol{A}} & \hat{\boldsymbol{b}} \\ 0 & 0 & \hat{\mathbb{I}} \end{bmatrix}$$

$$\hat{W}_{lower} = \begin{bmatrix} \hat{\mathbb{I}} & 0 & 0 \\ \hat{\boldsymbol{b}} & \hat{\boldsymbol{A}} & 0 \\ \hat{d} & \hat{\boldsymbol{c}} & \hat{\mathbb{I}} \end{bmatrix}$$

Where:
$\hat{b} \quad and \quad \hat{c}$ are operator-valued vectors and
$\hat{A}$ is an operator-valued matrix which is *not nessecarily either upper or lower triangular*.

In order to handle all combinations of upper and lower regular form in conjunction with left and right canonical (orhtogonal) forms we must extend ITensors *QR* decomposition capabilities to also include *QL*, *RQ* and *LQ* decomposition. Below and in the code these will be generically referred to as *QX* decomposition.

The V-block

For lower regular form the V-blocks are:

$$\hat{V}_L = \begin{bmatrix} \hat{\boldsymbol{A}}_L & 0 \\ \hat{\boldsymbol{c}}_L & \hat{\mathbb{I}} \end{bmatrix}$$

$$\hat{V}_R = \begin{bmatrix} \hat{\mathbb{I}} & 0 \\ \hat{\boldsymbol{b}}_L & \hat{\boldsymbol{A}}_L \end{bmatrix}$$

QX decomposition of the V-block

Block respecting *QL* decomposition is defined as *QL* decomposition of the of corresponding V-block:

1. Reshape

$$\hat{V}_{ab} \to V_{ab}^{mn} \to V_{(mna)b}$$

2. *QL* decompose

$$V_{(mna)b} = \sum_{k=1}^{\chi+1} Q_{(mna)k} L_{kb}$$

3. Reshape

$$Q_{(mna)k} \to \hat{Q}_{ak}$$

4. Stuff *Q* back into correct block of *W*.

5. Transfer *L* to the next site:

$$\hat{W}^{(i+1)} \to L\hat{W}^{(i+1)}$$

Fortunately, under the hood, ITensor takes care all the reshaping for us. After this process *W* will now be in left canonical form. Other cases and some additional details are described in tech notes.

`ITensorMPOCompression.block_qx` — Function

`block_qx(W::ITensor,ul::reg_form)::Tuple{ITensor,ITensor,Index}`

Perform a block respecting QX decomposition of the operator valued matrix `W`. The appropriate decomposition, QR, RQ, QL, LQ is selected based on the `reg_form ul` and the `dir` keyword argument. The new internal `Index` between Q and R/L is modified so that the tags are "Link,qx" instead "Link,qr" etc. returned by the qr/rq/ql/lq routines. Q and R are also gauge fixed so that the corner element of R is 1.0 and Q†Q=d𝕀 where d in the dimensionality of the local Hilbert space.

Arguments

- `W` Opertor valued matrix for decomposition.
- `ul` upper/lower regular form of `W`. We can auto detect here, but is more efficient if this is done by the higher level calling routines.

Keywords

- `orth::orth_type = right` : choose `left` or `right` orthogonal (canonical) form
- `epsrr::Float64 = 1e-14` : cutoff for rank revealing QX which removes zero pivot rows and columns. All rows with max(abs(R[:,j]))<epsrr are considered zero and removed. epsrr==0.0 indicates no rank reduction.

Returns a Tuple containing

- `Q` with orthonormal columns or rows depending on orth=left/right, dimensions: (χ+1)x(χ'+1)

- `R` or `L` depending on `ul` with dimensions: (χ+2)x(χ'+2)
- `iq` the new internal link index between `Q` and `R/L` with tags="Link,qx"

Example

```
julia>using ITensors
julia>using ITensorMPOCompression
julia>N=5; #5 sites
julia>NNN=2; #Include 2nd nearest neighbour interactions
julia>sites = siteinds("S=1/2",N);
#
#  Make a Hamiltonian directly, i.e. no using autoMPO
#
julia>H=make_transIsing_MPO(sites,NNN);
#
#  Use pprint to see the sructure for site #2. I = unit operator and S = any other
#
julia>pprint(H[2]) #H[1] is a row vector, so let's see what H[2] looks like
I 0 0 0 0
S 0 0 0 0
S 0 0 0 0
0 0 I 0 0
0 S 0 S I
#
#  Now do a block respecting QX decomposition. QL decomposition is chosen because
#  H[2] is in lower regular form and the default ortho direction if left.
#
julia>Q,L,iq=block_qx(H[2]); #Block respecting QL
#
#  The first column of Q is unchanged because it is outside the V-block.
#  Also one column was removed because rank revealing QX is the default algorithm.
#
julia>pprint(Q)
I 0 0 0
S 0 0 0
S 0 0 0
0 I 0 0
0 0 S I
#
#  Similarly L is missing one row due to rank revealing QX algorithm
#
julia>pprint(L,iq) #we need to tell pprint which index is the row index.
I 0 0 0 0
0 0 I 0 0
```

```
0 S 0 S 0
0 0 0 0 I
```

Orthogonalization (Canonical forms)

This is achieved by simply sweeping through the lattice and carrying out block respecting *QX* steps described above. For left canoncical form one starts at the left and sweeps right, and the converse applies for right canonical form.

`ITensors.orthogonalize!` — Function

```
orthogonalize!(H::MPO)
```

Bring an MPO into left or right canonical form using block respecting QR decomposition as described in:

> Daniel E. Parker, Xiangyu Cao, and Michael P. Zaletel Phys. Rev. B 102, 035147

Keywords

- `orth::orth_type = left` : choose `left` or `right` canonical form
- `sweeps::Int64` : number of sweeps to perform. If sweeps is zero or not set then sweeps continue until there is no change in the internal dimensions from rank revelaing QR.
- `epsrr::Float64 = 1e-14` : cutoff for rank revealing QX which removes zero pivot rows and columns. All rows with max(abs(R[:,j]))<epsrr are considered zero and removed.

Examples

```
julia> using ITensors
julia> using ITensorMPOCompression
julia> N=10; #10 sites
julia> NNN=7; #Include up to 7th nearest neighbour interactions
julia> sites = siteinds("S=1/2",N);
#
# This makes H directly, bypassing autoMPO.  (AutoMPO is too smart for this
# demo, it makes maximally reduced MPOs right out of the box!)
#
julia> H=make_transIsing_MPO(sites,NNN);
```

```
#
#  Make sure we have a regular form or orhtogonalize! won't work.
#
julia> is_lower_regular_form(H)==true
true
#
#  Let's see what the second site for the MPO looks like.
#  I = unit operator, and S = any other operator
#
julia> pprint(H[2])
I 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
S 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
.
.
.
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 I 0 0
0 S 0 S 0 0 S 0 0 0 S 0 0 0 0 S 0 0 0 0 0 S 0 0 0 0 0 0 S I
#
#  Now we can orthogonalize or bring it into canonical form.
#  Defaults are left orthogonal with rank reduction.
#
julia> orthogonalize!(H)
#
#  Wahoo .. rank reduction knocked the size of H way down, and we haven't
#  tried compressing yet!
#
julia> pprint(H[2])
I 0 0 0 0
S S S 0 0
0 0 0 S I
#
#  What do all the boond dimansions of H look like?  We will need compression (tru
#  in order to further bang down the size of H
#
julia> get_Dw(H)
9-element Vector{Int64}:
  3
  5
  9
 13
 12
  9
  6
  4
  3
#
```

```
#  wrap up with two more checks on the structure of H
#
julia> is_lower_regular_form(H)==true
true
julia> is_orthogonal(H,left)==true
true
```

Truncation (SVD compression)

Prior to truncation the MPO must first be rendered into canoncial form using the orthogonalize! function described above. If for example the MPO is right-lower canonical form then a truncation sweep starts do doing a block repsecting *QL* decomposition on site 1:

$$\hat{W}^1 \rightarrow \hat{Q}^1 L^1$$

We now must further factor *L* as follows

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathsf{L} & 0 \\ 0 & t & 1 \end{bmatrix} = ML' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathsf{M} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathbb{I} & 0 \\ 0 & t & 1 \end{bmatrix}$$

where internal sans-M matrix is what gets decomposed with SVD. Picking out this internal matrix is really the secret sauce for avoiding diverging singular values when the lattice size grows. After deomposition the *U* matrix is absorbed to the left sunch that W=QU which is left canoncial. *sV* gets combined with L' and transfered to next site to the right. There are many details to consider and these are explained in the technical notes.

`NDTensors.truncate!` — Function

```
truncate!(H::MPO)
```

Compress an MPO using block respecting SVD techniques as described in

> *Daniel E. Parker, Xiangyu Cao, and Michael P. Zaletel Phys. Rev. B 102, 035147*

Arguments

- `H` MPO for decomposition. If H is not already in the correct canonical form for compression, it will automatically be put into the correct form prior to compression.

Keywords

- `orth::orth_type = right` : choose `left` or `right` canonical form for the final output.
- `cutoff::Foat64 = 1e-14` : Using a `cutoff` allows the SVD algorithm to truncate as many states as possible while still ensuring a certain accuracy.
- `maxdim::Int64` : If the number of singular values exceeds `maxdim`, only the largest `maxdim` will be retained.
- `mindim::Int64` : At least `mindim` singular values will be retained, even if some fall below the cutoff

Example

```
julia> using ITensors
julia> using ITensorMPOCompression
julia> N=10; #10 sites
julia> NNN=7; #Include up to 7th nearest neighbour interactions
julia> sites = siteinds("S=1/2",N);
#
# This makes H directly, bypassing autoMPO.  (AutoMPO is too smart for this
# demo, it makes maximally reduced MPOs right out of the box!)
#
julia> H=make_transIsing_MPO(sites,NNN);
#
#  Make sure we have a regular form or truncate! won't work.
#
julia> is_lower_regular_form(H)==true
true

#
#  Now we can truncate with defaults of left orthogonal cutoff=1e-14.
#  truncate! returns the spectrum of singular values at each bond.  The largest si
#  values are remaining well under control.  i.e. no sign of divergences.
#
julia> truncate!(H)
9-element Vector{bond_spectrum}:
 bond_spectrum([0.307], 1)
 bond_spectrum([0.354, 0.035], 2)
 bond_spectrum([0.375, 0.045, 0.021], 3)
 bond_spectrum([0.385, 0.044, 0.026, 0.018], 4)
 bond_spectrum([0.388, 0.043, 0.031, 0.019, 0.001], 5)
 bond_spectrum([0.385, 0.044, 0.026, 0.018], 6)
```

```
 bond_spectrum([0.375, 0.045, 0.021], 7)
 bond_spectrum([0.354, 0.035], 8)
 bond_spectrum([0.307], 9)

julia> pprint(H[2])
I 0 0 0
S S S 0
0 S S I

#
#  We can see that bond dimensions have been drastically reduced.
#
julia> get_Dw(H)
9-element Vector{Int64}:
 3
 4
 5
 6
 7
 6
 5
 4
 3

julia> is_lower_regular_form(H)==true
true

julia> is_orthogonal(H,left)==true
true
```

Characterizations

The module has a number of functions for characterization of MPOs and operator-valued matrices. Some points to keep in mind:

1. An MPO must be in one of the regular forms in order for orthogonalization to work.
2. An MPO must be in one fo the orthononal (canonical) forms prior to SVD truncation (compression). However the truncate! function will detect this and do the orthogonalization if needed.
3. Lower (upper) regular form does not mean that the MPO is lower (upper) triangular. As explained in the Technical Notes the `A`-block does not need to be triangular.

4. Having said all that, most common hand constructed MPOs are either lower or upper triangular. Lower happens to be the more common convention.
5. The orthogonalize! operation just happens preserve lower (upper) trianglur form. However truncation (SVD) does not preserve triangular form.
6. At the moment autoMPO outputs (after a simple 2<–>Dw row & col swap `fix_autoMPO!`) lower *regular* form, but not lower *triangular* form (the `A` block is not triangular).

Regular forms

Regular forms are defined above in section Regular Forms

`ITensorMPOCompression.reg_form` — Type

```
@enum reg_form  upper lower
```

Indicates that an MPO or operator-valued matrix has either an `upper` or `lower` regular form. This becomes non-trival for rectangular matrices. See also `detect_regular_form` and related functions

`ITensorMPOCompression.detect_regular_form` — Function

```
detect_regular_form(W[,eps])::Tuple{Bool,Bool}
```

Inspect the structure of an operator-valued matrix W to see if it satisfies the regular form conditions as specified in Section III, definition 3 of

> *Daniel E. Parker, Xiangyu Cao, and Michael P. Zaletel Phys. Rev. B 102, 035147*

Arguments

- `W::ITensor` : operator-valued matrix to be characterized. W is expected to have 2 "Site" indices and 1 or 2 "Link" indices
- `eps::Float64 = 1e-14` : operators inside W with norm(W[i,j])<eps are assumed to be zero.

Returns a Tuple containing

- `reg_lower::Bool` Indicates W is in lower regular form.
- `reg_upper::Bool` Indicates W is in upper regular form.

The function returns two Bools in order to handle cases where W is not in regular form, returning (false,false) and W is in a special pseudo diagonal regular form, returning (true,true).

```
detect_regular_form(H[,eps])::Tuple{Bool,Bool}
```

Inspect the structure of an MPO `H` to see if it satisfies the regular form conditions.

Arguments

- `H::MPO` : MPO to be characterized.
- `eps::Float64 = 1e-14` : operators inside `W` with norm(W[i,j])<eps are assumed to be zero.

Returns a Tuple containing

- `reg_lower::Bool` Indicates all sites in `H` are in lower regular form.
- `reg_upper::Bool` Indicates all sites in `H` are in upper regular form.

The function returns two Bools in order to handle cases where `H` is not regular form, returning (`false,false`) and `H` is in a special pseudo-diagonal regular form, returning (`true,true`).

`ITensorMPOCompression.is_regular_form` — Function

```
is_regular_form(W,ul[,eps])::Bool
```

Determine is a operator-values matrix, `W`, is in `ul` regular form.

Arguments

- `W::ITensor` : operator-valued matrix to be characterized. `W` is expected to have 2 site indices and 1 or 2 link indices
- 'ul::reg_form' : choose `lower` or `upper`.
- `eps::Float64 = 1e-14` : operators inside `W` with norm(W[i,j])<eps are assumed to be zero.

Returns

- `true` if `W` is in `ul` regular form.

```
is_regular_form(H[,eps])::Bool
```

Determine is a MPO, `H`, is in either lower xor upper regular form. All sites in H must in the same (lower or upper) regular form in order to return true. In other words mixtures of lower and upper will fail.

Arguments

- `H::MPO` : MPO to be characterized.
- `eps::Float64 = 1e-14` : operators inside `H` with norm(W[i,j])<eps are assumed to be zero.

Returns

- `true` if *all* sites in `H` are in either lower xor upper regular form.

```
is_regular_form(H,ul[,eps])::Bool
```

Determine is a MPO, `H`, is in `ul` regular form. All sites in H must in the same `ul` regular form in order to return true.

Arguments

- `H::MPO` : MPO to be characterized.
- 'ul::reg_form' : choose `lower` or `upper`.
- `eps::Float64 = 1e-14` : operators inside `H` with norm(W[i,j])<eps are assumed to be zero.

Returns

- `true` if *all* sites in `H` are in `ul` regular form.

`ITensorMPOCompression.is_lower_regular_form` — Function

```
is_lower_regular_form(W[,eps])::Bool
```

Determine is a operator-values matrix, `W`, is in lower regular form.

Arguments

- `W::ITensor` : operator-valued matrix to be characterized. `W` is expected to have 2 site indices and 1 or 2 link indices
- `eps::Float64 = 1e-14` : operators inside `W` with norm(W[i,j])<eps are assumed to be zero.

Returns

- `true` if `W` is in lower regular form.

```
is_lower_regular_form(H[,eps])::Bool
```

Determine if all sites in an MPO, `H`, are in lower regular form.

Arguments

- `H::MPO` : MPO to be characterized.
- `eps::Float64 = 1e-14` : operators inside `H` with norm(W[i,j])<eps are assumed to be zero.

Returns

- `true` if *all* sites in `H` are in lower regular form.

`ITensorMPOCompression.is_upper_regular_form` — Function

```
is_upper_regular_form(W[,eps])::Bool
```

Determine is a operator-values matrix, `W`, is in upper regular form.

Arguments

- `W::ITensor` : operator-valued matrix to be characterized. `W` is expected to have 2 site indices and 1 or 2 link indices
- `eps::Float64 = 1e-14` : operators inside `W` with norm(W[i,j])<eps are assumed to be zero.

Returns

- `true` if `W` is in upper regular form.

```
is_upper_regular_form(H[,eps])::Bool
```

Determine if all sites in an MPO, `H`, are in upper regular form.

Arguments

- `H::MPO` : MPO to be characterized.
- `eps::Float64 = 1e-14` : operators inside `H` with norm(W[i,j])<eps are assumed to be zero.

Returns

- `true` if *all* sites in `H` are in upper regular form.

Orthogonal forms

The definition of orthogonal forms MPOs or operator-valued matrices are more complicated than those for MPSs. First we must defin inner product of two operator-valued matrices:

$$\sum \left\langle \hat{W}_{ab}^{\dagger}, \hat{W}_{ac} \right\rangle = \frac{\sum Tr\left[\hat{W}_{ab}\hat{W}_{ac}\right]}{Tr\left[\hat{\mathbb{I}}\right]} = \delta_{bc}$$

Where the summation limits depend on where the V-block is for `left/right` and `lower/upper`. The specifics are shown in table 6 in the Technical Notes

`ITensorMPOCompression.orth_type` — Type

```
@enum orth_type left right
```

Indicates that an MPO matrix satisfies the conditions for `left` or `right` canonical form

`ITensorMPOCompression.is_orthogonal` — Function

is_orthogonal(H,lr[,eps])::Bool

Test if all sites in an MPO statisfty the condition for `lr` orthogonal (canonical) form.

Arguments

- `H:MPO` : MPO to be characterized.
- `lr::orth_type` : choose `left` or `right` orthogonality condition to test for.
- `eps::Float64 = 1e-14` : operators inside H with norm(W[i,j])<eps are assumed to be zero.

Returns `true` of the MPO is in orthongal (canonical) form

Some utility functions

One of the difficult aspects of working with operator-valued matrices is that they have four indices and if one just does a naive @show W to see what's in there, you see a volumous output that is hard to read because of the default slicing selected by the @show overload. The pprint(W) (pretty print) function attempts solve this problem by simply showing you where the zero, unit and other operators reside.

`ITensorMPOCompression.pprint` — Function

pprint(W[,eps])

Show (pretty print) a schematic view of an operator-valued matrix. In order to display any `ITensor` as a matrix one must decide which index to use as the row index, and simiarly for the column index. `pprint` does this by inspecting the indices with "Site" tags to get the site number, and uses the "Link" indices `l=$n` as the column and `$(n-1)` as the row. The output symbols I,0,S have the obvious interpretations and S just means any (non zero) operator that is not I. This function can obviously be enchanced to recognize and display other symbols like X,Y,Z ... instead of just S.

Arguments

- `W::ITensor` : Operator-valued matrix for display.
- `eps::Float64 = 1e-14` : operators inside W with norm(W[i,j])<eps are assumed to be zero.

Examples

```
julia> pprint(W)
I 0 0 0 0
S 0 0 0 0
S 0 0 0 0
0 0 I 0 0
0 S 0 S I
```

```
pprint(H[,eps])
```

Display the structure of an MPO, one line for each lattice site. For each site the table lists

- n : lattice site number
- Dw1 : dimension of the row index (left link index)
- Dw2 : dimension of the column index (right link index)
- d : dimension of the local Hilbert space
- Reg. Form : U=upper, L=lower, D=diagonal, No = Not reg. form.
- Orth. Form : L=left, R=right, M=not orthogonal, B=both left and right.

- Tri. Form : U=upper triangular, L=lower triangular, F=full (not triangular), D=diagonal, R=row, C=column

Arguments

- `H::MPO` : MPO for display.
- `eps::Float64 = 1e-14` : operators inside H with norm(W[i,j])<eps are assumed to be zero.

Examples

```
julia> using ITensors
julia> using ITensorMPOCompression
julia> N=10; #10 sites
julia> NNN=7; #Include up to 7th nearest neighbour interactions
julia> sites = siteinds("S=1/2",N);
julia> H=make_transIsing_MPO(sites,NNN);
julia> pprint(H)
n    Dw1  Dw2   d   Reg.  Orth.  Tri.
                   Form  Form   Form
 1    1   30    2    L      M     R
 2   30   30    2    L      M     L
 3   30   30    2    L      M     L
 4   30   30    2    L      M     L
 5   30   30    2    L      M     L
 6   30   30    2    L      M     L
 7   30   30    2    L      M     L
 8   30   30    2    L      M     L
 9   30   30    2    L      M     L
10   30    1    2    L      M     C

julia> orthogonalize!(H,orth=right)
julia> pprint(H)
n    Dw1  Dw2   d   Reg.  Orth.  Tri.
                   Form  Form   Form
 1    1    3    2    L      M     R
 2    3    4    2    L      R     L
 3    4    5    2    L      R     L
 4    5    6    2    L      R     L
 5    6    7    2    L      R     L
 6    7    6    2    L      R     L
 7    6    5    2    L      R     L
 8    5    4    2    L      R     L
 9    4    3    2    L      R     L
10    3    1    2    L      R     C
julia> truncate!(H,orth=left)
```

```
julia> pprint(H)
n    Dw1  Dw2   d   Reg.  Orth.  Tri.
                    Form  Form   Form
 1    1    3    2    L      L     R
 2    3    4    2    L      L     L
 3    4    5    2    L      L     F
 4    5    6    2    L      L     F
 5    6    7    2    L      L     F
 6    7    6    2    L      L     F
 7    6    5    2    L      L     F
 8    5    4    2    L      L     F
 9    4    3    2    L      L     L
10    3    1    2    L      M     C
```

Test Hamiltonians

For development and demo purposes it is useful to have a quick way to make Hamiltonian MPOs for various models. Right now we have three Hamiltonians available

1. Direct transverse Ising model with arbitrary neighbour interactions
2. autoMPO transverse Ising model with arbitrary neighbour interactions
3. autoMPO Heisenberg model with arbitrary neighbour interactions

The autoMPO MPOs come pre-truncated to they are not useful testing truncation.

`ITensorMPOCompression.make_transIsing_MPO` — Function

```
make_transIsing_MPO(sites[,NNN=1[,hx=0.0[,ul=lower[,J=1.0]]]])
```

Directly coded build up a transverse Ising model Hamiltonian with up NNN neighbour interactions. The interactions are hard coded to decay like J/(i-j) between sites `i` and `j`.

Arguments

- `sites` : Site set defining the lattice of sites.
- `NNN::Int64=1` : Number of nearest neighbour interactions to include in H
- `hx::Float64=0.0` : External magnetic field in `x` direction.
- `ul::reg_form=lower` : build H with lower or upper regular form.
- `J::Float64=1.0` : Nearest neighbour interaction strength.

`ITensorMPOCompression.make_transIsing_AutoMPO` — Function

make_transIsing_AutoMPO(sites,NNN[,hx=0.0[,J=1.0]])

Use `ITensor.autoMPO` to build up a transverse Ising model Hamiltonian with up NNN neighbour interactions. The interactions are hard coded to decay like J/(i-j) between sites `i` and `j`.

Arguments

- `sites` : Site set defining the lattice of sites.
- `NNN::Int64` : Number of nearest neighbour interactions to include in H
- `hx::Float64=0.0` : External magnetic field in `x` direction.
- `J::Float64=1.0` : Nearest neighbour interaction strength.

`ITensorMPOCompression.make_Heisenberg_AutoMPO` — Function

```
make_Heisenberg_AutoMPO(sites,[NNN=1[,hz=0.0[,J=1.0]]])
```

Use `ITensor.autoMPO` to build up a Heisenberg model Hamiltonian with up NNN neighbour interactions. The interactions are hard coded to decay like J/(i-j) between sites `i` and `j`.

Arguments

- `sites` : Site set defining the lattice of sites.
- `NNN::Int64` : Number of nearest neighbour interactions to include in H
- `hz::Float64=0.0` : External magnetic field in `z` direction.
- `J::Float64=1.0` : Nearest neighbour interaction strength.

`ITensorMPOCompression.fix_autoMPO!` — Method

```
fix_autoMPO!(H::MPO)
```

Convert AutoMPO output into lower regular form. If Dw x Dw' are the dimensions of the Link indices for any site then this routine simply does 2 swaps:

1. Swap row 2 with row Dw
2. Swap column 2 with column Dw'

Arguments

- `H::MPO` : MPO to be characterized.

MPO is modified in place into lower regular form

Examples

```
julia> using ITensors
julia> using ITensorMPOCompression
julia> N=5
julia> sites = siteinds("S=1/2",5);
julia> ampo = OpSum();
julia> for j=1:N-1
julia>     add!(ampo,1.0, "Sz", j, "Sz", j+1);
julia>     add!(ampo,0.5, "S+", j, "S-", j+1);
julia>     add!(ampo,0.5, "S-", j, "S+", j+1);
julia> end
julia> H=MPO(ampo,sites);
julia> pprint(H[2])
I 0 0 0 0
0 I S S S
S 0 0 0 0
S 0 0 0 0
S 0 0 0 0

julia> fix_autoMPO!(H)
julia> pprint(H[2])
I 0 0 0 0
S 0 0 0 0
S 0 0 0 0
S 0 0 0 0
0 S S S I
```