ITensor Notes

September 25, 2023

Install ITensor

[1]:
```
# using Pkg
# Pkg.add("ITensors")
```

Load ITensor

[2]:
```
using ITensors
```

Index() and Itensor() functions

- **Index()**: Define an Index represents a single tensor index with fixed dimension dim.
- **Itensor()**: Constructor for an ITensor from a TensorStorage and a set of indices.

[3]:
```
# ?ITensor #This is how you ask for help in Julia
```

[4]:
```
# ?Index #This is how you ask for help in Julia
```

Example of usage:

[5]:
```
i = Index(3)
j = Index(3)

A = ITensor(i,j)

A[i=>1, j => 2] = 2 #This is how we set the elements inside the tensor (it does␣
 ↪not matter the order).

println(A)
```

```
ITensor ord=2
Dim 1: (dim=3|id=723)
Dim 2: (dim=3|id=637)
NDTensors.Dense{Int64, Vector{Int64}}
 3×3
 0  2  0
 0  0  0
 0  0  0
```

[6]:
```
A[i=>1,j=>2] #This is how we can get the elements inside the tensor  (it does␣
 ↪not matter the order).
```

[6]: 2

Matrix Example:

Let's play a little bith with this. Consider the tensor product: $A_{ij} * B_{ik}$

[7]:
```
i = Index(3)
j = Index(3)
k = Index(3)

A = ITensor(i,j)
B = ITensor(i,k)

for j_index in 1:dim(j)
    A[i=>1, j => j_index] = j_index
    A[i=>2, j => j_index] = 3 + j_index
    A[i=>3, j => j_index] = 6 + j_index
end

for k_index in 1:dim(j)
    B[i=>1, k => k_index] = (k_index)/10
    B[i=>2, k => k_index] = (3 + k_index)/10
    B[i=>3, k => k_index] = (6 + k_index)/10
end
```

[8]:
```
println(A)
```

```
ITensor ord=2
Dim 1: (dim=3|id=701)
Dim 2: (dim=3|id=146)
NDTensors.Dense{Int64, Vector{Int64}}
 3×3
 1  2  3
 4  5  6
 7  8  9
```

[9]:
```
println(B)
```

```
ITensor ord=2
Dim 1: (dim=3|id=701)
Dim 2: (dim=3|id=970)
NDTensors.Dense{Float64, Vector{Float64}}
 3×3
 0.1  0.2  0.3
 0.4  0.5  0.6
 0.7  0.8  0.9
```

I choose this values just because all of them are different. Let's try to contract the indixes using Itensor

[10]:
```
C = A*B #Use * is how ITensor do the contraction of the common indexes between A and B.

println(C)
```

```
ITensor ord=2
Dim 1: (dim=3|id=146)
Dim 2: (dim=3|id=970)
NDTensors.Dense{Float64, Vector{Float64}}
 3×3
 6.6   7.800000000000001   9.0
 7.8   9.3                10.8
 9.0  10.8                12.6
```

Let's try exactly the same without using Itensor, just Julia:

[11]:
```
m, n = 3,3
A_matrix = fill(0.0, (3,3))
B_matrix = fill(0.0, (3,3))

for j_index in 1:3, i_index in 1:3
        A_matrix[i_index, j_index] = A[i=>i_index,j=>j_index]
end

for k_index in 1:3, i_index in 1:3
        B_matrix[i_index, k_index] = B[i=>i_index,k=>k_index]
end
```

[12]:
```
A_matrix #i,j
```

[12]:
```
3×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0
```

[13]:
```
B_matrix #k,i
```

[13]:
```
3×3 Matrix{Float64}:
 0.1  0.2  0.3
 0.4  0.5  0.6
 0.7  0.8  0.9
```

Here we do not have an index, just matrixes. We must be careful thinking what index we can to contract, and do the operation that we really want.

It is not just A_matrix*B_matrix

[14]:
```
A_matrix*B_matrix
```

[14]:
```
3×3 Matrix{Float64}:
  3.0   3.6   4.2
  6.6   8.1   9.6
 10.2  12.6  15.0
```

We want $A_{ij} * B_{ik}$, and $(A * B)_{ik} = A_{ij} * B_{jk}$, that is very different.

So $A_{ij} * B_{ik} = A^T_{ji} * B_{ik} = (A^T * B)_{jk}$

[15]:
```
transpose(A_matrix)*B_matrix
```

[15]:
```
3×3 Matrix{Float64}:
 6.6   7.8   9.0
 7.8   9.3  10.8
 9.0  10.8  12.6
```

We got the same result, but was harder without ITensor.

randomITensor() function Create an ITensor with normally-distributed random elements instead of specific values.

[16]:
```
A = randomITensor(i,j,k)

println(A)
```

```
ITensor ord=3
Dim 1: (dim=3|id=701)
Dim 2: (dim=3|id=146)
Dim 3: (dim=3|id=970)
NDTensors.Dense{Float64, Vector{Float64}}
 3×3×3
[:, :, 1] =
 -0.8448438054962486   -0.5766655883899466   0.7801098827611991
 -0.30089353376558714  -2.2241772069346366  -0.9005492667931966
 -0.5834833980583248    0.4584843686427207  -2.277873757673437

[:, :, 2] =
 -1.3811354216516143    0.5842736743114801    1.1944366406803495
  1.8389793857318693    0.15337753288940334   0.6484452260327175
  0.24476498535588453  -0.19578279379412858  -0.5989647370527114

[:, :, 3] =
 0.4923653977929339    0.6179540958176872   0.060175853610214804
 0.9275529715256585    0.3844983581247267  -0.6214343662268764
 0.045967213876599994  1.5557077324808857   1.803525162565913
```

Linear combinations of ITensors ITensors may also be subtracted and multiplied by scalars, including complex scalars, for example:

[17]:
```
A = randomITensor(i,j,k)
B = randomITensor(k,i,j)
```

[17]:
```
ITensor ord=3 (dim=3|id=970) (dim=3|id=701) (dim=3|id=146)
NDTensors.Dense{Float64, Vector{Float64}}
```

[18]:
```
C = 4*A - B/2
D = A + 3.0im * B
```

[18]:
```
ITensor ord=3 (dim=3|id=701) (dim=3|id=146) (dim=3|id=970)
NDTensors.Dense{ComplexF64, Vector{ComplexF64}}
```

[19]:
```
println(D)
```

```
ITensor ord=3
Dim 1: (dim=3|id=701)
Dim 2: (dim=3|id=146)
Dim 3: (dim=3|id=970)
NDTensors.Dense{ComplexF64, Vector{ComplexF64}}
 3×3×3
[:, :, 1] =
 -1.2636815759290825 - 0.3433407044362427im   0.6350604112667522 +
2.733662732604917im     0.8010109105099674 + 4.619186631734982im
 -0.2572559434748985 - 3.281362147412273im   -0.0658340351734703 +
1.4375275591100594im   -0.7933623934214925 - 2.291518252472905im
   -1.69517206129063 - 3.8930059352071607im  -0.6552433101291986 -
4.556829192424722im   -0.04866912860821715 - 2.66817319504562im

[:, :, 2] =
   2.065923628326572 - 2.096633651917678im   -1.433848845571969 +
4.163077843976798im     0.8297364186611381 + 5.709457290756246im
  0.1330465848784986 + 2.395517679110611im   0.3895452987625015 +
2.279160888958553im    0.27557233096224243 - 5.025239950307328im
 -0.5363720043390507 - 0.68464231376605 43im  0.5735315695592704 -
1.5561895174145108im  -0.09718128364392055 - 2.5032404334402893im

[:, :, 3] =
 0.14411050254379662 - 0.026589962751263 27im  -0.5575814141691695 +
1.2731655637578747im   0.18914274630118327 + 0.8357659498974666im
  -2.571647603611888 + 0.1251498665004337im    1.2739291728898343 -
0.23073585401782887im  -0.4658178884559792 + 3.1132149060140017im
 -1.7405268892119272 + 4.096599405408341im    -0.5397212760232711 +
0.8583513664479483im    0.3763582067421386 + 7.021678279524817im
```

This is just possible because A and B have the same indexes:

[20]:
```
l = Index(3)
```

```
A = randomITensor(i,j,k)
B = randomITensor(k,i,l)
```

[20]:
```
ITensor ord=3 (dim=3|id=970) (dim=3|id=701) (dim=3|id=126)
NDTensors.Dense{Float64, Vector{Float64}}
```

[21]:
```
C = 4*A - B/2
```

```
You are trying to add an ITensor with indices:

((dim=3|id=970), (dim=3|id=701), (dim=3|id=126))

into an ITensor with indices:

((dim=3|id=701), (dim=3|id=146), (dim=3|id=970))

but the indices are not permutations of each other.


Stacktrace:
 [1] error(s::String)
   @ Base .\error.jl:35
 [2] _map!!(f::Function, R::NDTensors.DenseTensor{Float64, 3,␣
 ↪Tuple{Index{Int64}, Index{Int64}, Index{Int64}}, NDTensors.Dense{Float64,␣
 ↪Vector{Float64}}}, T1::NDTensors.DenseTensor{Float64, 3, Tuple{Index{Int64},␣
 ↪Index{Int64}, Index{Int64}}, NDTensors.Dense{Float64, Vector{Float64}}}, T2::
 ↪NDTensors.DenseTensor{Float64, 3, Tuple{Index{Int64}, Index{Int64},␣
 ↪Index{Int64}}, NDTensors.Dense{Float64, Vector{Float64}}})
   @ ITensors C:\Users\joaqu\.julia\packages\ITensors\yZbTa\src\itensor.jl:1929
 [3] map!(f::Function, R::ITensor, T1::ITensor, T2::ITensor)
   @ ITensors C:\Users\joaqu\.julia\packages\ITensors\yZbTa\src\itensor.jl:1955
 [4] copyto!
   @ C:\Users\joaqu\.julia\packages\ITensors\yZbTa\src\broadcast.jl:321 [inline
 ]
 [5] materialize!
   @ .\broadcast.jl:884 [inlined]
 [6] materialize!
   @ .\broadcast.jl:881 [inlined]
 [7] -(A::ITensor, B::ITensor)
   @ ITensors C:\Users\joaqu\.julia\packages\ITensors\yZbTa\src\itensor.jl:1872
 [8] top-level scope
   @ In[21]:1
```

prime(), delta(), combiner() and dag() functions We already see that we can use * to contract the common indexes between two or more tensors. However, * can be used in different ways if we also use the functions prime(), delta(), combiner() and dag().

Consider two Tensors:

A = randomITensor(i,j) B = randomITensor(i,j)

They have two common indixes:

[22]:
```
commoninds(A,B) #It returns the id of the common indixes between A and B
```

[22]:
```
2-element Vector{Index{Int64}}:
 (dim=3|id=701)
 (dim=3|id=970)
```

As we expected, these indixes are i and j:

[23]:
```
i,j
```

[23]:
```
((dim=3|id=701), (dim=3|id=146))
```

Then if we use * between A and B, the operator will contract both indexes:

[24]:
```
C = A*B #We got a tensor of range zero (an scalar)
```

[24]:
```
ITensor ord=2 (dim=3|id=146) (dim=3|id=126)
NDTensors.Dense{Float64, Vector{Float64}}
```

This has sense because it is just $A_{i,j} * B_{i,j}$. IF we contract i and j, then we must have and scalar. We can access to this scalar in two ways:

[25]:
```
C[], scalar(C)
```

```
DimensionMismatch: In scalar(T) or T[], ITensor T is not a scalar (it has␣
 ↪indices ((dim=3|id=146), (dim=3|id=126))).

Stacktrace:
 [1] getindex(T::ITensor)
   @ ITensors C:\Users\joaqu\.julia\packages\ITensors\yZbTa\src\itensor.jl:1145
 [2] top-level scope
   @ In[25]:1
```

There are some cases when we do not want to contract all the common indexes. For example, if we just want to contract j even if there are two common indexes we can use the function **prime()**:

[26]:
```
A_prime = prime(A,i)
```

[26]:
```
ITensor ord=3 (dim=3|id=701)' (dim=3|id=146) (dim=3|id=970)
NDTensors.Dense{Float64, Vector{Float64}}
```

The ITensor A_prime has the same elements as A but has indices (i',j) instead of (i,j).

[27]:
```
println(A)

println(A_prime)
```

```
ITensor ord=3
Dim 1: (dim=3|id=701)
Dim 2: (dim=3|id=146)
Dim 3: (dim=3|id=970)
NDTensors.Dense{Float64, Vector{Float64}}
 3×3×3
[:, :, 1] =
 -0.298300139746288    0.7354383564643635   0.4822169212168984
  0.8441801089665497  -1.322201408473126    1.3742715358406357
 -0.584990273365746   -0.4105820488507161  -0.8392737158803838

[:, :, 2] =
 -1.8083800888223183   1.4490873973049743   1.260428902013258
 -0.7848888887060964  -1.0926160511327083  -0.6073094306435473
  0.2557463018607807  -1.7030575777391446  -2.300219617337377

[:, :, 3] =
 -0.08646451575128565  -0.8640184553605545    1.1837326650915048
  2.0155602586289656   -1.1598628352255387   -1.1754289405094986
  0.7088522333980871   -0.31536023087832754   1.175236286441898
ITensor ord=3
Dim 1: (dim=3|id=701)'
Dim 2: (dim=3|id=146)
Dim 3: (dim=3|id=970)
NDTensors.Dense{Float64, Vector{Float64}}
 3×3×3
[:, :, 1] =
 -0.298300139746288    0.7354383564643635   0.4822169212168984
  0.8441801089665497  -1.322201408473126    1.3742715358406357
 -0.584990273365746   -0.4105820488507161  -0.8392737158803838

[:, :, 2] =
 -1.8083800888223183   1.4490873973049743   1.260428902013258
 -0.7848888887060964  -1.0926160511327083  -0.6073094306435473
  0.2557463018607807  -1.7030575777391446  -2.300219617337377

[:, :, 3] =
 -0.08646451575128565  -0.8640184553605545    1.1837326650915048
  2.0155602586289656   -1.1598628352255387   -1.1754289405094986
  0.7088522333980871   -0.31536023087832754   1.175236286441898
```

Note: We also can do A_prime = prime(A,i,j) or just A_prime = A'. Then A_prime will have the same elements as A but has indices (i',j') instead of (i,j).

In this case A_prime and B just have one common index:

[28]:
```
commoninds(A_prime,B)
```

[28]:
```
1-element Vector{Index{Int64}}:
 (dim=3|id=970)
```

So,

[29]:
```
C = A_prime*B

println(C)
```

```
ITensor ord=4
Dim 1: (dim=3|id=701)'
Dim 2: (dim=3|id=146)
Dim 3: (dim=3|id=701)
Dim 4: (dim=3|id=126)
NDTensors.Dense{Float64, Vector{Float64}}
 3×3×3×3
[:, :, 1, 1] =
 -1.1773959146979005  -0.004047790231941573   2.299663486875412
  2.375207038374048   -2.50787055040364      -1.223576972395286
  0.7741095375521281  -1.4449492197646876    -0.1059788302556881

[:, :, 2, 1] =
  3.478483379624589   -1.894816340084601   -3.6180450219929554
 -0.735983033774696    3.3975225128217095   2.1463465255286014
 -1.1230703641278277   3.5343326431921946   3.1648745390399196

[:, :, 3, 1] =
 -2.7609719596305022     2.7062650796740395   2.282663792583736
 -0.027384643749334428  -2.9738086163170316   0.6243315973348044
 -0.25862057371665487   -2.7540425457115014  -3.947455907863056

[:, :, 1, 2] =
 -1.0098048076401673   1.81380902813124    -0.5143368414040522
 -2.6692120943303386   0.5996915408212641   0.8861240520426468
 -0.604455948380298   -0.6966080109223389  -2.6701595736565222

[:, :, 2, 2] =
 -0.8944000806064448   1.44282097420243    -0.588373416448392
 -2.7856053498423567   0.9918493206597007   0.256614684143062
 -0.3160834155350881  -0.5618956944553082  -2.224232090350268

[:, :, 3, 2] =
 0.864812689820933   -1.42933477705063    -0.36174478531488624
 0.4661597391677142   0.9300101097275109  -1.2540976580321044
 0.6172879429747129   0.8179522065398602   1.9484021307325774

[:, :, 1, 3] =
 0.4830695686018389  -1.8397798531727574    0.9270599700619051
```

```
 2.256394466371158   -0.46931880176454066  -2.1030753346488
 1.158729661441091    0.23213798389588663   2.490452197474153

[:, :, 2, 3] =
 -0.09206970146019315  -1.020317958812225    -0.6457865658827678
 -2.435603266303851     2.6705562446925697   -3.385180893468011
  1.4263640587421953    0.22976877551634828   0.8693553511556846

[:, :, 3, 3] =
  1.112867429287792   -0.046240849518563165  -0.6729825732487928
  1.277734555849658   -0.5489073065432487     2.52359295749246
 -1.1183328647147461   0.9334477946637199     0.5387503225676937
```

Instead of doing $A_{i,j} * B_{i,j}$, we did $A_{i',j} * B_{i,j} = C_{i',i}$. Now if we want to get the same scalar as before, we just need to contract when $i = i'$. We can do this using the function **delta()**:

[30]:
```
println(delta(i,i'))
```

```
ITensor ord=2
Dim 1: (dim=3|id=701)
Dim 2: (dim=3|id=701)'
NDTensors.Diag{Float64, Float64}
 3×3
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

Then,

[31]:
```
println(C*delta(i,i'))
```

```
ITensor ord=2
Dim 1: (dim=3|id=146)
Dim 2: (dim=3|id=126)
NDTensors.Dense{Float64, Vector{Float64}}
 3×3
 -2.1719995221892514   -3.1781222145078116  -3.070866562416758
  0.6394321768782665    3.623610555330801    1.7642241861835322
  0.49855410454095717   1.6906799734715872  -1.9193706008384126
```

Note: delta() output is the identity tensor, we can build it with more indexes:

[32]:
```
println(delta(i,j,k))
```

```
ITensor ord=3
Dim 1: (dim=3|id=701)
Dim 2: (dim=3|id=146)
Dim 3: (dim=3|id=970)
NDTensors.Diag{Float64, Float64}
 3×3×3
[:, :, 1] =
```

```
 1.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0

[:, :, 2] =
 0.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  0.0

[:, :, 3] =
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  1.0
```

Note: Julia allows us to use directly the symbol instead of delta.

[33]:
```
println( (k,i,j))
```

```
ITensor ord=3
Dim 1: (dim=3|id=970)
Dim 2: (dim=3|id=701)
Dim 3: (dim=3|id=146)
NDTensors.Diag{Float64, Float64}
 3×3×3
[:, :, 1] =
 1.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0

[:, :, 2] =
 0.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  0.0

[:, :, 3] =
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  1.0
```

In addition to this, we also can use the Itensor product operator (*) to reshape Tensors using the functions **combiner()** and **dag()**.

As an example consider this tensor:

[34]:
```
A = randomITensor(i,j,k,l)

println(A)
```

```
ITensor ord=4
Dim 1: (dim=3|id=701)
```

```
Dim 2: (dim=3|id=146)
Dim 3: (dim=3|id=970)
Dim 4: (dim=3|id=126)
NDTensors.Dense{Float64, Vector{Float64}}
 3×3×3×3
[:, :, 1, 1] =
 -0.16467305983041852   -1.3799829327081825    0.3641270485862605
  0.8724462033111595     1.0214697788154985    1.242192323190504
  0.016985316198691227  -0.15572247294502684  -0.4674855966542696

[:, :, 2, 1] =
  0.5118804463690205   -0.8958436849056645   0.22165949526336484
 -0.12913085992471535   1.7998749629616766  -0.613701386594952
  1.367350184841786     0.6512655055772542  -0.6048185333995241

[:, :, 3, 1] =
  0.517520910157864    -0.9375686807827592  -0.1311100132699169
 -0.9971536117207376    0.6009958590291659  -1.0750848472719128
  0.08895613730104732   2.1751757489310024   0.3472960605565142

[:, :, 1, 2] =
 -1.3113630132812824   0.8979562793257395   -0.2793031615940889
  0.3212316104998843  -0.29135758471152484  -0.8805960579942903
 -0.6169398706507204  -0.31332297744293     -0.5528229144686562

[:, :, 2, 2] =
 -1.0577890735334174  -0.15727043594286308  -0.3989102379264582
  0.583655931667914   -0.26106428550926747  -0.42116288488162296
  0.2549206570016759   1.3910164963830485   -0.07019247880814687

[:, :, 3, 2] =
 -1.4242679443916104  -0.8301653805348318     0.13025225933947215
  0.5577293593358994  -0.040869441291267815   0.7204207520011389
  0.8230945573734006   0.6439207763216788    -1.190454110145767

[:, :, 1, 3] =
 -0.22127075172922372  -0.4263310336243007   0.9658811244451502
  1.312910436940747    -0.16292651787746673  0.16187298884291448
  1.6029302738743811    1.9913516788175174   0.9908219335783708

[:, :, 2, 3] =
  0.4113962013040699    0.28602719155228296  -0.8960486490477458
  0.31873229922446294   0.21176299821949984  -0.8180847504266139
 -0.28780931627937445  -0.5558401850646877   -0.48487140711674526

[:, :, 3, 3] =
 -0.7873387958767754   0.2770473867162359  -0.09665414388229573
  0.511219995794985    0.1361343222576142   0.009407614030854244
```

```
 1.1745203808373812  -0.5836328397420842   0.19439349449147614
```

We have a tensor of order 4. If we want to reshape it as a tensor of order 2, we can use the function combiner to merge i, j and k indexes:

[35]:
```
Mix_i_j_k = combiner(i,j,k, tags = "ijk") #tags is an optional parameter when␣
↪we define indexes, is just a label to know what means the index. In this␣
↪case is useful to remember which indexes was merged.
```

[35]:
```
ITensor ord=4 (dim=27|id=294|"ijk") (dim=3|id=701) (dim=3|id=146) (dim=3|id=970)
NDTensors.Combiner
```

Then we just need to do the product:

[36]:
```
A_reshaped_ijk_l = Mix_i_j_k*A

println(A_reshaped_ijk_l)
```

```
ITensor ord=2
Dim 1: (dim=27|id=294|"ijk")
Dim 2: (dim=3|id=126)
NDTensors.Dense{Float64, Vector{Float64}}
 27×3
 -0.16467305983041852   -1.3113630132812824    -0.22127075172922372
  0.8724462033111595     0.3212316104998843     1.312910436940747
  0.016985316198691227  -0.6169398706507204     1.6029302738743811
 -1.3799829327081825     0.8979562793257395    -0.4263310336243007
  1.0214697788154985    -0.29135758471152484   -0.16292651787746673
 -0.15572247294502684   -0.31332297744293      1.9913516788175174
  0.3641270485862605    -0.2793031615940889    0.9658811244451502
  1.242192323190504     -0.8805960579942903    0.16187298884291448
 -0.4674855966542696    -0.5528229144686562    0.9908219335783708
  0.5118804463690205    -1.0577890735334174    0.4113962013040699
 -0.12913085992471535    0.583655931667914     0.31873229922446294
  1.367350184841786      0.2549206570016759   -0.28780931627937445
 -0.8958436849056645    -0.15727043594286308   0.28602719155228296
  1.7998749629616766    -0.26106428550926747   0.21176299821949984
  0.6512655055772542     1.3910164963830485   -0.5558401850646877
  0.22165949526336484   -0.3989102379264582   -0.8960486490477458
 -0.613701386594952     -0.42116288488162296  -0.8180847504266139
 -0.6048185333995241    -0.07019247880814687  -0.48487140711674526
  0.517520910157864     -1.4242679443916104   -0.7873387958767754
 -0.9971536117207376     0.5577293593358994    0.511219995794985
  0.08895613730104732    0.8230945573734006    1.1745203808373812
 -0.9375686807827592    -0.8301653805348318    0.2770473867162359
  0.6009958590291659    -0.040869441291267815  0.1361343222576142
  2.1751757489310024     0.6439207763216788   -0.5836328397420842
 -0.1311100132699169     0.13025225933947215  -0.09665414388229573
 -1.0750848472719128     0.7204207520011389    0.009407614030854244
```

```
 0.3472960605565142    -1.190454110145767      0.19439349449147614
```

We also could thing in a mix i,j and k,l in order to have a matrix of dimensions ij x kl. If we want to do that we just need to use two combiners:

[37]:
```
Mix_i_j = combiner(i,j, tags = "ij")
Mix_k_l = combiner(k,l, tags = "kl")

A_reshaped_ij_kl = Mix_i_j*(Mix_k_l*A)

println(A_reshaped_ij_kl)
```

```
ITensor ord=2
Dim 1: (dim=9|id=77|"ij")
Dim 2: (dim=9|id=848|"kl")
NDTensors.Dense{Float64, Vector{Float64}}
 9×9
 -0.16467305983041852    0.5118804463690205    0.517520910157864
-1.3113630132812824   -1.0577890735334174   -1.4242679443916104
-0.22127075172922372   0.4113962013040699   -0.7873387958767754
  0.8724462033111595    -0.12913085992471535  -0.9971536117207376
0.3212316104998843    0.583655931667914     0.5577293593358994
1.312910436940747     0.31873229922446294   0.511219995794985
  0.016985316198691227   1.367350184841786     0.08895613730104732
-0.6169398706507204    0.2549206570016759    0.8230945573734006
1.6029302738743811   -0.28780931627937445   1.1745203808373812
 -1.3799829327081825    -0.8958436849056645   -0.9375686807827592
0.8979562793257395   -0.15727043594286308  -0.8301653805348318
-0.4263310336243007    0.28602719155228296   0.2770473867162359
  1.0214697788154985     1.7998749629616766    0.6009958590291659
-0.29135758471152484  -0.26106428550926747  -0.040869441291267815
-0.16292651787746673   0.21176299821949984   0.1361343222576142
 -0.15572247294502684    0.6512655055772542    2.1751757489310024
-0.31332297744293      1.3910164963830485    0.6439207763216788
1.9913516788175174   -0.5558401850646877   -0.5836328397420842
  0.3641270485862605     0.22165949526336484  -0.1311100132699169
-0.2793031615940889   -0.3989102379264582    0.13025225933947215
0.9658811244451502   -0.8960486490477458   -0.09665414388229573
  1.242192323190504     -0.613701386594952    -1.0750848472719128
-0.8805960579942903   -0.42116288488162296   0.7204207520011389
0.16187298884291448  -0.8180847504266139    0.009407614030854244
 -0.4674855966542696    -0.6048185333995241    0.3472960605565142
-0.5528229144686562   -0.07019247880814687  -1.190454110145767
0.9908219335783708   -0.48487140711674526   0.19439349449147614
```

Now, if we just want to recover the initial shape we can use the function **dag()**:

[38]:
```
A_reconstructed_1 = (dag(Mix_i_j_k)*A_reshaped_ijk_l)
A_reconstructed_2 = dag(Mix_k_l)*(dag(Mix_i_j)*A_reshaped_ij_kl)
```

```
# Let's see if these tensor are the same as A:

# println(A_reconstructed_1, A_reconstructed_2, A)
A == A_reconstructed_1 == A_reconstructed_2
```

[38]: true

Descomposition of ITensors (QR and SVD): QR and SVD are two very useful and famous ways to descompose Matrixes (2-rank tensors). These descompositions just exist for Matrixes, so if we have a tensor of rank i x j x k x l is necessary to reshape the tensor as a matrix (could be something like ij x kl or ijk x l or i x jkl, etc.), apply the algorithm to build the QR or the SVD descomposition, and finally reshape again to recover the structure i x j x k x l. **qr()** and **svd()** do this large process for us.

We just need to specify which index(es) we want for Q (for QR) or U (For SVD) matrix at the end of the process.

[39]:
```
# Q,R = qr(A_reshaped_ij_kl)
```

[40]:
```
# U,S,V = svd(A_reshaped_ij_kl)
```

To see this, just consider the following tensor:

[41]:
```
i = Index(3, "i")
j = Index(3, "j")
k = Index(3, "k")

A = randomITensor(i,j,k)
```

[41]:
```
ITensor ord=3 (dim=3|id=50|"i") (dim=3|id=650|"j") (dim=3|id=461|"k")
NDTensors.Dense{Float64, Vector{Float64}}
```

[42]:
```
Q,R = qr(A,(i,j))

println(Q) #Has i,j
println(R) #Has k
```

```
ITensor ord=3
Dim 1: (dim=3|id=50|"i")
Dim 2: (dim=3|id=650|"j")
Dim 3: (dim=3|id=593|"Link,qr")
NDTensors.Dense{Float64, Vector{Float64}}
 3×3×3
[:, :, 1] =
 -0.1902172670369029  -0.43525092002584426   0.528261504935396
 -0.2611504781035302   0.267454250345887    -0.22972206929178257
  0.3393359711040933   0.4250570198082059    0.08359343040087201
```

```
[:, :, 2] =
 -0.07683140633112742  0.2554062925885529   -0.28138016366276813
 -0.7190499936685935   0.23140371777732213  -0.22185223172274554
 -0.29540127043290704  0.17156245103770795  -0.3364449320597507

[:, :, 3] =
 -0.38959165745387536   0.18456179163861552   0.5479388934315914
  0.13361103080971884  -0.13377314414835023  -0.28238469207503925
 -0.20780943791276663  -0.39986698332685344  -0.44198598593872385
ITensor ord=2
Dim 1: (dim=3|id=593|"Link,qr")
Dim 2: (dim=3|id=461|"k")
NDTensors.Dense{Float64, Vector{Float64}}
 3×3
 4.280488316259302   1.2089882744268212   0.5758125863817085
 0.0                -2.7298958701800315   0.6085520284241868
 0.0                 0.0                 -2.4613637232923726
```

or

[43]:
```
Q,R = qr(A,(i))

println(Q) #Has i
println(R) #Has j,k
```

```
ITensor ord=2
Dim 1: (dim=3|id=50|"i")
Dim 2: (dim=3|id=489|"Link,qr")
NDTensors.Dense{Float64, Vector{Float64}}
 3×3
 -0.40597746599625406   0.5212964520832154  0.7506212801064942
 -0.5573690075339631   -0.7921477257431893  0.24868005557442255
  0.7242389706056042   -0.3174145391209179  0.6121486125205138
ITensor ord=3
Dim 1: (dim=3|id=489|"Link,qr")
Dim 2: (dim=3|id=650|"j")
Dim 3: (dim=3|id=461|"k")
NDTensors.Dense{Float64, Vector{Float64}}
 3×3×3
[:, :, 1] =
 2.0055861649960427   1.4359934533612928  -0.1107819714899394
 0.0                 -2.4556190423190607   1.8441237982586838
 0.0                  0.0                  1.6718244682983783

[:, :, 2] =
 -0.028729595236745405   0.7015439963623904   -0.015513466438785484
 -1.7015619378313918    -0.40796757464748923   0.15000167395441666
  1.1392241986518825    -0.9671491578821215    1.7616167951051338
```

```
[:, :, 3] =
 0.5669137851513223   0.8409067921550986   0.9296041622920099
 0.9773536890656648  -1.204039716663662   -1.268211712949225
 0.6971619285790955   0.5589798917231826  -0.23609662668132148
```

In any case,

[44]:
```
A   Q*R
```

[44]:
```
true
```

With SVD descomposition is exactly the same:

[45]:
```
U,S,V = svd(A,(i,j))

println(U) # Has i,j
println(S)
println(V) # Has k
```

```
ITensor ord=3
Dim 1: (dim=3|id=50|"i")
Dim 2: (dim=3|id=650|"j")
Dim 3: (dim=3|id=830|"Link,u")
NDTensors.Dense{Float64, Vector{Float64}}
 3×3×3
[:, :, 1] =
 -0.14109223665338408  -0.4916461352472188    0.5351903132463541
 -0.11150772392407438   0.22115379945032237  -0.15651609930774182
  0.4083273780511589    0.4065627538835162    0.18461744689994486

[:, :, 2] =
 -0.14103071087050909  -0.03959231986648622  0.4760315114811087
  0.6882946780454582   -0.2986228607815069   0.042937468598669815
  0.07064917666350777  -0.42682146438121904  0.00033946944660978584

[:, :, 3] =
 -0.3925183468106545   0.21320252933324307   0.3813625226983017
 -0.3419076437316273   0.06992921229312914  -0.3941964492562095
 -0.271795917031121   -0.1501030104539572   -0.530519508351712
ITensor ord=2
Dim 1: (dim=3|id=830|"Link,u")
Dim 2: (dim=3|id=768|"Link,v")
NDTensors.Diag{Float64, Vector{Float64}}
 3×3
 4.5675645292988225  0.0                 0.0
 0.0                 2.903638135621514   0.0
 0.0                 0.0                 2.1686427247682
ITensor ord=2
Dim 1: (dim=3|id=461|"k")
```

```
Dim 2: (dim=3|id=768|"Link,v")
NDTensors.Dense{Float64, Vector{Float64}}
 3×3
 0.9133241895390596   -0.19366788830903056   0.3582341047984411
 0.38480153517621757   0.6983521301318021   -0.6035164296573878
 0.13329179768029858  -0.6890551874692588   -0.7123455940011555
```

[46]:
```
U,S,V = svd(A,(i))

println(U) # Has i
println(S)
println(V) # Has j,k
```

```
ITensor ord=2
Dim 1: (dim=3|id=50|"i")
Dim 2: (dim=3|id=1|"Link,u")
NDTensors.Dense{Float64, Vector{Float64}}
 3×3
 -0.7745260941106948  -0.5758999517355441   -0.2616267859616619
  0.4049392262398443  -0.13368632875548267  -0.9045176551931734
  0.4859357494503854  -0.8065154748018653    0.3367480308943371
ITensor ord=2
Dim 1: (dim=3|id=1|"Link,u")
Dim 2: (dim=3|id=365|"Link,v")
NDTensors.Diag{Float64, Vector{Float64}}
 3×3
 4.44753072216934  0.0                0.0
 0.0               2.961428227304642  0.0
 0.0               0.0                2.3337062260527865
ITensor ord=3
Dim 1: (dim=3|id=650|"j")
Dim 2: (dim=3|id=461|"k")
Dim 3: (dim=3|id=365|"Link,v")
NDTensors.Dense{Float64, Vector{Float64}}
 3×3×3
[:, :, 1] =
  0.1987188116235778   0.28643028824006267  -0.1656593937339516
  0.6274794628265924   0.18996029157262845   0.2981949028992104
 -0.4442194746799167  -0.10374307560348922   0.3524150010629198

[:, :, 2] =
 -0.1867782802930767   -0.4017724970485623  -0.25825602381047097
 -0.18488147699572066   0.2394371719720492  -0.28445111151431013
 -0.4927912834068433   -0.566035515077954   -0.03651510318459808

[:, :, 3] =
 0.7341427278501184   -0.4606072338252436   0.3414281847958379
 0.02768420000615268   0.26324578004852006  0.012112852150782366
```

```
0.17925697188372813  -0.13768983674447208  0.10487705171990147
```

In any case,

[47]:
```
U*S*V   A # true
```

[47]:
```
true
```

MPS and MPO examples

[48]:
```
function heisenberg_mpo(N) # Make N S=1/2 spin indices
    sites = siteinds("S=1/2",N)
    # Input the operator terms

    os = OpSum()

    for i=1:N-1
        os += "Sz",i,"Sz",i+1
        os += 1/2,"S+",i,"S-",i+1
        os += 1/2,"S-",i,"S+",i+1
    end

    # Convert these terms to an MPO
    H = MPO(os,sites)
    return H
end
```

[48]:
```
heisenberg_mpo (generic function with 1 method)
```

[ ]:
```
H = heisenberg_mpo(10)
```

[ ]:
```
H[2]
```

[ ]:
```
println(H[2])
```

[ ]:
```

```