Tensors and the `TensorMap` type

Types

```
abstract type AbstractTensorMap{S<:IndexSpace, N₁, N₂} end
const AbstractTensor{S<:IndexSpace, N} = AbstractTensorMap{S, N, 0}

struct TensorMap{S<:IndexSpace, N₁, N₂, I<:Sector, A<:Union{<:DenseMatrix,SectorDict{I
    data::A
    codom::ProductSpace{S,N₁}
    dom::ProductSpace{S,N₂}
    rowr::SectorDict{I,FusionTreeDict{F₁,UnitRange{Int}}}
    colr::SectorDict{I,FusionTreeDict{F₂,UnitRange{Int}}}
    function TensorMap{S, N₁, N₂, I, A, F₁, F₂}(data::A,
                codom::ProductSpace{S,N₁}, dom::ProductSpace{S,N₂},
                rowr::SectorDict{I,FusionTreeDict{F₁,UnitRange{Int}}},
                colr::SectorDict{I,FusionTreeDict{F₂,UnitRange{Int}}}) where
                    {S<:IndexSpace, N₁, N₂, I<:Sector, A<:SectorDict{I,<:DenseMatrix},
                     F₁<:FusionTree{I,N₁}, F₂<:FusionTree{I,N₂}}
        eltype(valtype(data)) ⊆ field(S) ||
            @warn("eltype(data) = $(eltype(data)) ⊈ $(field(S)))", maxlog=1)
        new{S, N₁, N₂, I, A, F₁, F₂}(data, codom, dom, rowr, colr)
    end
    function TensorMap{S, N₁, N₂, Trivial, A, Nothing, Nothing}(data::A,
                codom::ProductSpace{S,N₁}, dom::ProductSpace{S,N₂}) where
                    {S<:IndexSpace, N₁, N₂, A<:DenseMatrix}
        eltype(data) ⊆ field(S) ||
            @warn("eltype(data) = $(eltype(data)) ⊈ $(field(S)))", maxlog=1)
        new{S, N₁, N₂, Trivial, A, Nothing, Nothing}(data, codom, dom)
    end
end

const Tensor{S<:IndexSpace, N, I<:Sector, A, F₁, F₂} = TensorMap{S, N, 0, I, A, F₁, F₂
const TrivialTensorMap{S<:IndexSpace, N₁, N₂, A<:DenseMatrix} = TensorMap{S, N₁, N₂, T

struct TensorKeyIterator{I<:Sector, F₁<:FusionTree{I}, F₂<:FusionTree{I}}
    rowr::SectorDict{I, FusionTreeDict{F₁, UnitRange{Int}}}
    colr::SectorDict{I, FusionTreeDict{F₂, UnitRange{Int}}}
end
struct TensorPairIterator{I<:Sector, F₁<:FusionTree{I}, F₂<:FusionTree{I}, A<:DenseMat
    rowr::SectorDict{I, FusionTreeDict{F₁, UnitRange{Int}}}
    rowr::SectorDict{I, FusionTreeDict{F₁, UnitRange{Int}}}
    colr::SectorDict{I, FusionTreeDict{F₂, UnitRange{Int}}}
    data::SectorDict{I, A}
end

const TensorIterator{I<:Sector, F₁<:FusionTree{I}, F₂<:FusionTree{I}} = Union{TensorKe

struct AdjointTensorMap{S<:IndexSpace, N₁, N₂, I<:Sector, A, F₁, F₂} <: AbstractTensor
    parent::TensorMap{S, N₂, N₁, I, A, F₂, F₁}
end

const AdjointTrivialTensorMap{S<:IndexSpace, N₁, N₂, A<:DenseMatrix} =
    AdjointTensorMap{S, N₁, N₂, Trivial, A, Nothing, Nothing}

const EuclideanTensorSpace = TensorSpace{<:EuclideanSpace}
const EuclideanTensorMapSpace = TensorMapSpace{<:EuclideanSpace}
const AbstractEuclideanTensorMap = AbstractTensorMap{<:EuclideanTensorSpace}
const EuclideanTensorMap = TensorMap{<:EuclideanTensorSpace}
[Q]: Could we define the `AbstractTensorMap{<:EuclideanTensorSpace}`? EuclideanTensorS
```

Properties

On both instances and types:

```
storagetype(t::AbstractTensorMap) # gives the way the tensor data are stored, now all 
similarstoragetype(t::AbstractTensorMap, T)
numout(t::AbstractTensorMap) # gives N_1 for the codomain
numin(t::AbstractTensorMap) # gives N_2 for the domain
numind(t::AbstractTensorMap) # gives N_1+N_2
const order = numind
codomainind(t::AbstractTensorMap) # 1:N_1
domainind(t::AbstractTensorMap) # N_1+1:N_1+N_2
allind(t::AbstractTensorMap) # 1:N_1+N_2
```

On instances:

```
codomian(t::AbstractTensorMap)
codomain(t::AbstractTensorMap, i) # `i`th index space of the codomain of the tensor ma
domain(t::AbstractTensorMap)
domain(t::AbstractTensorMap, i) # `i`th index space of the domain of the tensor map `t
source(t::AbstractTensorMap) # gives domain
target(t::AbstractTensorMap) # gives codomain
space(t::AbstractTensorMap) # give HomSpace
space(t::AbstractTensorMap, i::Int) # `i`th index space of the HomSpace corresponding 
adjointtensorindex(t::AbstractTensorMap{<:IndexSpace, N₁, N₂}, i) # gives the index in
adjointtensorindex(t::AbstractTensorMap{<:IndexSpace, N₁, N₂}, I) # gives the index in
adjointtensorindices(t::AbstractTensorMap, indices::IndexTuple)
tensormaptype(::Type{S}, N₁::Int, N₂::Int, ::Type{T}) where {S,T} # Return the correct
blocksectors(t::TensorMap) # Return an iterator over the different unique coupled sect
hasblock(t::TensorMap, s::Sector) # Check whether the sector `s` is in the block secto
blocks(t::TensorMap) # Return the data of the tensor map as a `SingletonDict` (for tri
block(t::TensorMap, s::Sector) # Return the data of tensor map corresponding to the bl
fusiontrees(t::TensorMap) # Return tbe TensorKeyIterator for all possible splitting an
Base.getindex(t::TensorMap{<:IndexSpace,N₁,N₂,I}, f1::FusionTree{I,N₁}, f2::FusionTree
Base.getindex(t::TensorMap{<:IndexSpace,N₁,N₂,I}, sectors::Tuple{Vararg{I}}) # `sector
```

Constructors

```
TensorMap(f, codom::ProductSpace{S,N₁}, dom::ProductSpace{S,N₂}) where {S<:IndexSpace,
TensorMap(data::DenseArray, codom::ProductSpace{S,N₁}, dom::ProductSpace{S,N₂}; tol =
TensorMap(data::AbstractDict{<:Sector,<:DenseMatrix}, codom::ProductSpace{S,N₁}, dom::
TensorMap(f,::Type{T}, codom::ProductSpace{S}, dom::ProductSpace{S}) where {S<:IndexSp
TensorMap(::Type{T}, codom::ProductSpace{S}, dom::ProductSpace{S}) where {S<:IndexSpac
TensorMap(::UndefInitializer, ::Type{T}, codom::ProductSpace{S}, dom::ProductSpace{S})
TensorMap(::UndefInitializer, codom::ProductSpace{S}, dom::ProductSpace{S}) where {S<:
TensorMap(::Type{T}, codom::TensorSpace{S}, dom::TensorSpace{S}) where {T<:Number, S<:
TensorMap(dataorf, codom::TensorSpace{S}, dom::TensorSpace{S}) where {S<:IndexSpace}
TensorMap(dataorf, ::Type{T}, codom::TensorSpace{S}, dom::TensorSpace{S}) where {T<:Nu
TensorMap(codom::TensorSpace{S}, dom::TensorSpace{S}) where {S<:IndexSpace}
TensorMap(dataorf, T::Type{<:Number}, P::TensorMapSpace{S}) where {S<:IndexSpace}
TensorMap(dataorf, P::TensorMapSpace{S}) where {S<:IndexSpace}
TensorMap(T::Type{<:Number}, P::TensorMapSpace{S}) where {S<:IndexSpace}
TensorMap(P::TensorMapSpace{S}) where {S<:IndexSpace}
Tensor(dataorf, T::Type{<:Number}, P::TensorSpace{S}) where {S<:IndexSpace}
Tensor(dataorf, P::TensorSpace{S}) where {S<:IndexSpace}
Tensor(T::Type{<:Number}, P::TensorSpace{S}) where {S<:IndexSpace}
Tensor(P::TensorSpace{S}) where {S<:IndexSpace}
Base.adjoint(t::TensorMap) = AdjointTensorMap(t)
Base.adjoint(t::AdjointTensorMap) = t.parent
zero(t::AbstractTensorMap) # Creat a tensor that is similar to the tensor map `t` with
one!(t::AbstractTensorMap) # Overwrite the tensor map `t` by a tensor map in which eve
one(t::AbstractTensorMap) # Creat a tensor map that similar to tensor map `t` and with
id([A::Type{<:DenseMatrix} = Matrix{Float64},] space::VectorSpace) # Construct the ide
isomorphism([A::Type{<:DenseMatrix} = Matrix{Float64},] cod::VectorSpace, dom::VectorS
unitary([A::Type{<:DenseMatrix} = Matrix{Float64},] cod::VectorSpace, dom::VectorSpace
isometry([A::Type{<:DenseMatrix} = Matrix{Float64},] cod::VectorSpace, dom::VectorSpac
```

Linear Operations

```
copy!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap)
copy(t::AbstractTensorMap)
fill!(t::AbstractTensorMap, value::Number)
adjoint!(tdst::AbstractEuclideanTensorMap, tsrc::AbstractEuclideanTensorMap)
mul!(t1::AbstractTensorMap, t2::AbstractTensorMap, α::Number)
mul!(t1::AbstractTensorMap, α::Number, t2::AbstractTensorMap)
mul!(tC::AbstractTensorMap, tA::AbstractTensorMap, tB::AbstractTensorMap, α = true, β
-(t::AbstractTensorMap)
*(t::AbstractTensorMap, α::Number)
*(α::Number, t::AbstractTensorMap)
*(t1::AbstractTensorMap, t2::AbstractTensorMap)
rmul!(t::AbstractTensorMap, α::Number) = mul!(t, t, α)
lmul!(α::Number, t::AbstractTensorMap) = mul!(t, α, t)
axpy!(α::Number, t1::AbstractTensorMap, t2::AbstractTensorMap)
+(t1::AbstractTensorMap, t2::AbstractTensorMap)
-(t1::AbstractTensorMap, t2::AbstractTensorMap)
axpby!(α::Number, t1::AbstractTensorMap, β::Number, t2::AbstractTensorMap)
exp!(t::TensorMap)
exp(t::AbstractTensorMap)
inv(t::AbstractTensorMap)
^(t::AbstractTensorMap, p::Integer)
pinv(t::AbstractTensorMap; kwargs...)
Base.:(\)(t1::AbstractTensorMap, t2::AbstractTensorMap)
/(t1::AbstractTensorMap, t2::AbstractTensorMap)
/(t::AbstractTensorMap, α::Number)
Base.:\(α::Number, t::AbstractTensorMap)
:cos, :sin, :tan, :cot, :cosh, :sinh, :tanh, :coth, :atan, :acot, :asinh
:sqrt, :log, :asin, :acos, :acosh, :atanh, :acoth
dot(t1::AbstractEuclideanTensorMap, t2::AbstractEuclideanTensorMap) # Return the eleme
norm(t::AbstractEuclideanTensorMap, p::Real = 2) # Return the norm of the tensor map `
normalize!(t::AbstractTensorMap, p::Real = 2) # Replace the tensor map `t` with the no
normalize(t::AbstractTensorMap, p::Real = 2) #  Creat a new tensor map that is similar
tr(t::AbstractTensorMap) # Return the trace of the true block diagonal matrix that rep
sylvester(A::AbstractTensorMap, B::AbstractTensorMap, C::AbstractTensorMap) # it compu
catdomain(t1::AbstractTensorMap{S, N₁, 1}, t2::AbstractTensorMap{S, N₁, 1}) where {S,
catcodomain(t1::AbstractTensorMap{S, 1, N₂}, t2::AbstractTensorMap{S, 1, N₂}) where {S
⊗(t1::AbstractTensorMap{S}, t2::AbstractTensorMap{S}, ...) # results in a new `TensorM
⊠(t1::AbstractTensorMap{<:EuclideanSpace{ℂ}}, t2::AbstractTensorMap{<:EuclideanSpace{ℂ
```

Index manipulations

```

```

General arguments

This last page explains how to create and manipulate tensors in TensorXD.jl. As this is probably the most important part of the manual, we will also focus more strongly on the usage and interface, and less so on the underlying implementation. The only aspect of the implementation that we will address is the storage of the tensor data, as this is important to know how to create and initialize a tensor, but will in fact also shed light on how some of the methods work.

As mentioned, all tensors in TensorXD.jl are interpreted as linear maps (morphisms) from a domain (a `ProductSpace{S,N₂}`) to a codomain (another `ProductSpace{S,N₁}`), with the same `S<:ElementarySpace` that labels the type of spaces associated with the individual tensor indices. The overall type for all such tensor maps is `AbstractTensorMap{S, N₁, N₂}`. Note that we place information about the codomain before that of the domain. Indeed, we have already encountered the constructor for the concrete parametric type `TensorMap` in the form `TensorMap(..., codomain, domain)`. This convention is opposite to the mathematical notation, e.g. $\mathrm{Hom}(W, V)$ or $f : W \to V$, but originates from the fact that a normal matrix is also denoted as having size `m × n` or is constructed in Julia as `Array(..., (m, n))`, where the first integer `m` refers to the codomain being `m`- dimensional, and the seond integer `n` to the domain being `n`-dimensional. This also explains why we have consistently used the symbol W for spaces in the domain and V for spaces in the codomain. A tensor map $t : (W_1 \otimes \ldots \otimes W_{N_2}) \to (V_1 \otimes \ldots \otimes V_{N_1})$ will be created in Julia as `TensorMap(..., V1 ⊗ ... ⊗ VN₁, W1 ⊗ ... ⊗ WN₂)`.

Furthermore, the abstract type `AbstractTensor{S,N}` is just a synonym for `AbstractTensorMap{S,N,0}`, i.e. for tensor maps with an empty domain, which is equivalent to the unit of the monoidal category, or thus, the field of scalars $\Bbbk$.

Currently, `AbstractTensorMap` has two subtypes. `TensorMap` provides the actual implementation, where the data of the tensor is stored in a `DenseArray` (more specifically a `DenseMatrix` as will be explained below). `AdjointTensorMap` is a simple wrapper type to denote the adjoint of an existing `TensorMap` object. In the future, additional types could be defined, to deal with sparse data, static data, diagonal data, etc...

Storage of tensor data

Before discussion how to construct and initalize a `TensorMap{S}`, let us discuss what is meant by 'tensor data' and how it can efficiently and compactly be stored. Let us first discuss the case `sectortype(S) == Trivial` sector, i.e. the case of no symmetries. In that case the data of a tensor `t = TensorMap(..., V1 ⊗ ... ⊗ VN₁, W1 ⊗ ... ⊗ WN₂)` can just be represented as a multidimensional array of size

```
(dim(V1), dim(V2), …, dim(VN₁), dim(W1), …, dim(WN₂))
```

which can also be reshaped into matrix of size

```
(dim(V1)*dim(V2)*…*dim(VN₁), dim(W1)*dim(W2)*…*dim(WN₂))
```

and is really the matrix representation of the linear map that the tensor represents. In particular, given a second tensor `t2` whose domain matches with the codomain of `t`, function composition amounts to multiplication of their corresponding data matrices. Similarly, tensor factorizations such as the singular value decomposition, which we discuss below, can act directly on this matrix representation.

> **Note**
>
> One might wonder if it would not have been more natural to represent the tensor data as `(dim(V1), dim(V2), …, dim(VN₁), dim(WN₂), …, dim(W1))` given how employing the duality naturally reverses the tensor product, as encountered with the interface of `repartition` for fusion trees. However, such a representation, when plainly `reshape`d to a matrix, would not have the above properties and would thus not constitute the matrix representation of the tensor in a compatible basis.

Now consider the case where `sectortype(S) == I` for some `I` which has `FusionStyle(I) == UniqueFusion()`, i.e. the representations of an Abelian group, e.g. `I == Irrep[ℤ₂]` or `I == Irrep[U₁]`. In this case, the tensor data is associated with sectors `(a1, a2, …, aN₁) ∈ sectors(V1 ⊗ V2 ⊗ … ⊗ VN₁)` and `(b1, …, bN₂) ∈ sectors(W1 ⊗ … ⊗ WN₂)` such that they fuse to a same common charge, i.e. `(c = first(⊗(a1, …, aN₁))) == first(⊗(b1, …, bN₂))`. The data associated with this takes the form of a multidimensional array with size `(dim(V1, a1), …, dim(VN₁, aN₁), dim(W1, b1), …, dim(WN₂, bN₂))`, or equivalently, a matrix of with row size `dim(V1, a1)*…*dim(VN₁, aN₁) == dim(codomain, (a1, …, aN₁))` and column size `dim(W1, b1)*…*dim(WN₂, aN₂) == dim(domain, (b1, …, bN₂))`.

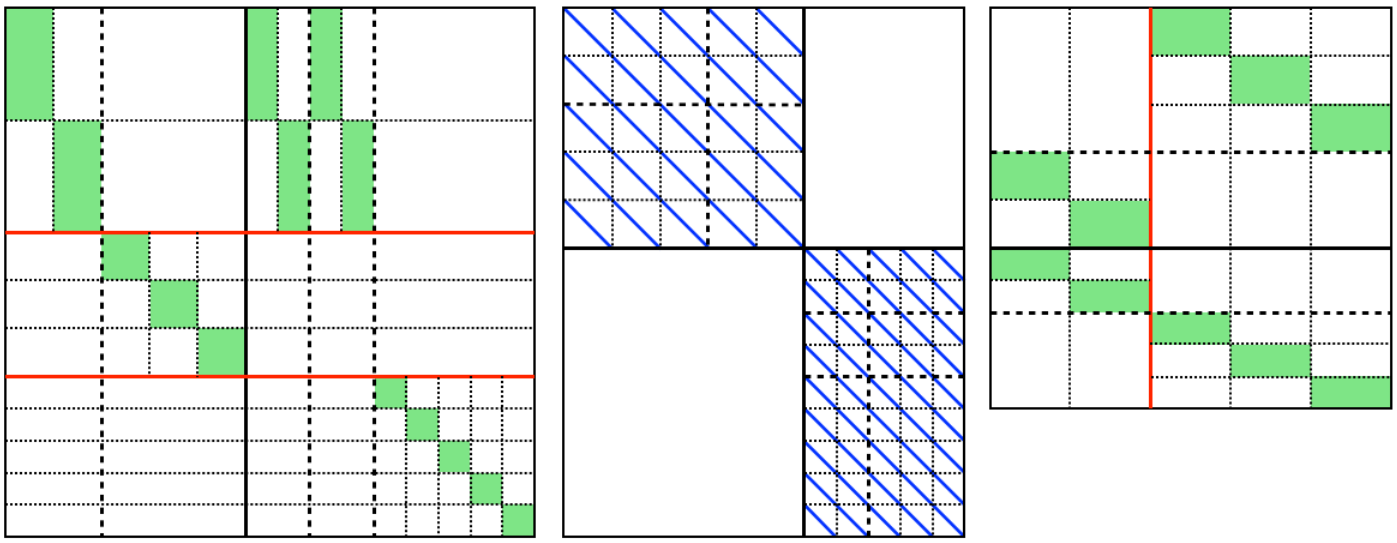
However, there are multiple combinations of `(a1, …, aN₁)` giving rise to the same `c`, and so there is data associated with all of these, as well as all possible combinations of `(b1, …, bN₂)`. Stacking all matrices for different `(a1,…)` and a fixed value of `(b1,…)` underneath each other, and for fixed value of `(a1,…)` and different values of `(b1,…)` next to each other, gives rise to a larger block matrix of all data associated with the central sector `c`. The size of this matrix is exactly `(blockdim(codomain, c), blockdim(domain, c))` and these matrices are exactly the diagonal blocks whose existence is guaranteed by Schur's lemma, and which are labeled by the coupled sector `c`. Indeed, if we would represent the tensor map `t` as a matrix without explicitly using the symmetries, we could reorder the rows and columns to group data corresponding to sectors that fuse to the same `c`, and the resulting block diagonal representation would emerge. This basis transform is thus a permutation, which is a unitary operation, that will cancel or go through trivially for linear algebra operations such as composing tensor maps (matrix multiplication) or tensor factorizations such as a singular value decomposition. For such linear algebra operations, we can thus directly act on these diagonal blocks that emerge after a basis transform, provided that the partition of the tensor indices in domain and codomain of the tensor are in line with our needs. For example, composing two tensor maps amounts to multiplying the matrices corresponding to the same `c` (provided that its subblocks labeled by the different combinations of sectors are ordered in the same way, which we guarantee by associating a canonical order with sectors). Henceforth, we refer to the `blocks` of a tensor map as those diagonal blocks, the existence of which is provided by Schur's lemma and which are labeled by the coupled sectors `c`. We directly store these blocks as `DenseMatrix` and gather them as values in a dictionary, together with the corresponding coupled sector `c` as key. For a given tensor `t`, we can access a specific block as `block(t, c)`, whereas `blocks(t)` yields an iterator over pairs `c=>block(t,c)`.

The subblocks corresponding to a particular combination of sectors then correspond to a particular view for some range of the rows and some range of the colums, i.e. `view(block(t, c), m₁:m₂, n₁:n₂)` where the ranges `m₁:m₂` associated with `(a1, …, aN₁)` and `n₁:n₂` associated with `(b₁, …, bN₂)` are stored within the fields of the instance `t` of type `TensorMap`. This `view` can then lazily be reshaped to a multidimensional array, for which we rely on the package Strided.jl. Indeed, the data in this `view` is not contiguous, because the stride between the different columns is larger than the length of the columns. Nonetheless, this does not pose a problem and even as multidimensional array there is still a definite stride associated with each dimension.

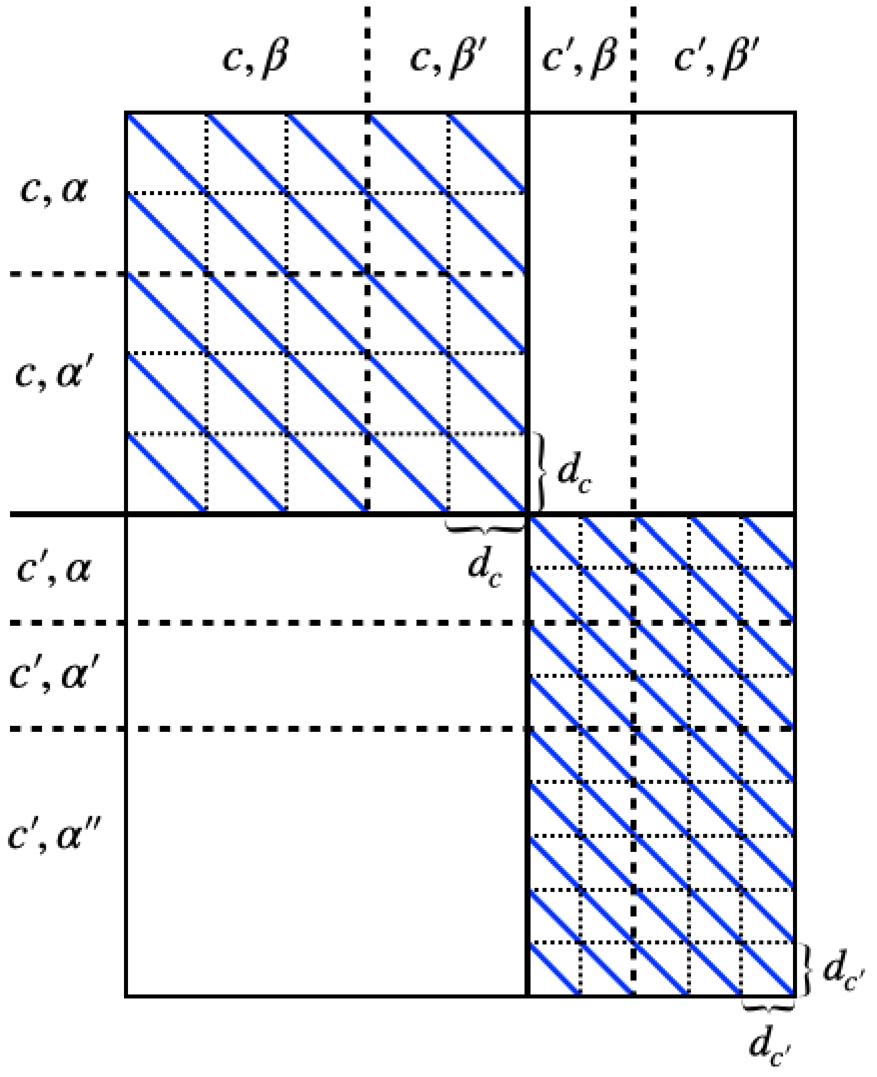
When `FusionStyle(I) isa MultipleFusion`, things become slightly more complicated. Not only do `(a1, …, aN₁)` give rise to different coupled sectors `c`, there can be multiply ways in which they fuse to `c`. These different possibilities are enumerated by the iterator `fusiontrees((a1, …, aN₁), c)` and `fusiontrees((b1, …, bN₂), c)`, and with each of those, there is tensor data that takes the form of a multidimensional array, or, after reshaping, a matrix of size `(dim(codomain, (a1, …, aN₁)), dim(domain, (b1, …, bN₂))))`. Again, we can stack all such matrices with the same value of `f₁ ∈ fusiontrees((a1, …, aN₁), c)` horizontally (as they all have the same number of rows), and with the same value of `f₂ ∈ fusiontrees((b1, …, bN₂), c)` vertically (as they have the same number of columns). What emerges is a large matrix of size `(blockdim(codomain, c), blockdim(domain, c))` containing all the tensor data associated with the coupled sector `c`, where `blockdim(P, c) = sum(dim(P, s)*length(fusiontrees(s, c)) for s in sectors(P))` for some instance `P` of `ProductSpace`. The tensor implementation does not distinguish between abelian or non-abelian sectors and still stores these matrices as a `DenseMatrix`, accessible via `block(t, c)`.

At first sight, it might now be less clear what the relevance of this block is in relation to the full matrix representation of the tensor map, where the symmetry is not exploited. The essential interpretation is still the same. Schur's lemma now tells that there is a unitary basis transform which makes this matrix representation block diagonal, more specifically, of the form $\bigoplus_{c} B_c \otimes \mathbb{1}_c$, where B_c denotes `block(t,c)` and $\mathbb{1}_c$ is an identity matrix of size `(dim(c), dim(c))`. The reason for this extra identity is that the group representation is recoupled to act as $\bigoplus_{c} \mathbb{1} \otimes u_c(g)$ for all $g \in \mathrm{I}$, with $u_c(g)$ the matrix representation of group element g according to the irrep c. In the abelian case, `dim(c) == 1`, i.e. all irreducible representations are one-dimensional and Schur's lemma only dictates that all off-diagonal blocks are zero. However, in the non-Abelian case the basis transform to the block diagonal representation is not simply a permutation matrix, but a more general unitary matrix composed of the different fusion trees. Indeed, let us denote the fusion trees `f₁ ∈ fusiontrees((a1, …, aN₁), c)` as $X^{a_1,\ldots,a_{N_1}}_{c,\alpha}$ where $\alpha = (e_1, \ldots, e_{N_1-2}; \mu_1, \ldots, \mu_{N_1-1})$ is a collective label for the internal sectors `e` and the vertex degeneracy labels μ of a generic fusion tree, as discussed in the corresponding section. The tensor is then represented as

$$
\begin{aligned}
t &= \bigoplus_{a_1\ldots a_{N_1}} \bigoplus_{b_1\ldots b_{N_2}} \bigoplus_{c,\alpha,\beta} t^{c}_{(a_1\ldots a_{N_1})\alpha,(b_1\ldots b_{N_2})\beta} \otimes \left(X^{a_1\ldots a_{N_1}}_{c,\alpha} \circ (X^{b_1\ldots b_{N_2}}_{c,\beta})^\dagger \right) \\
&= \bigoplus_{a_1\ldots a_{N_1}} \bigoplus_{b_1\ldots b_{N_2}} \bigoplus_{c,\alpha,\beta} \left(\mathbb{1}_{n_{a_1}\times\ldots\times n_{a_{N_1}}} \otimes X^{a_1\ldots a_{N_1}}_{c,\alpha} \right) \circ \left(t^{c}_{(a_1\ldots a_{N_1})\alpha,(b_1\ldots b_{N_2})\beta} \otimes \mathbb{1}_c \right) \circ \left(\mathbb{1}_{n_{b_1}\times\ldots\times n_{b_{N_2}}} \otimes X^{b_1\ldots b_{N_2}}_{c,\beta} \right)^\dagger
\end{aligned}
$$

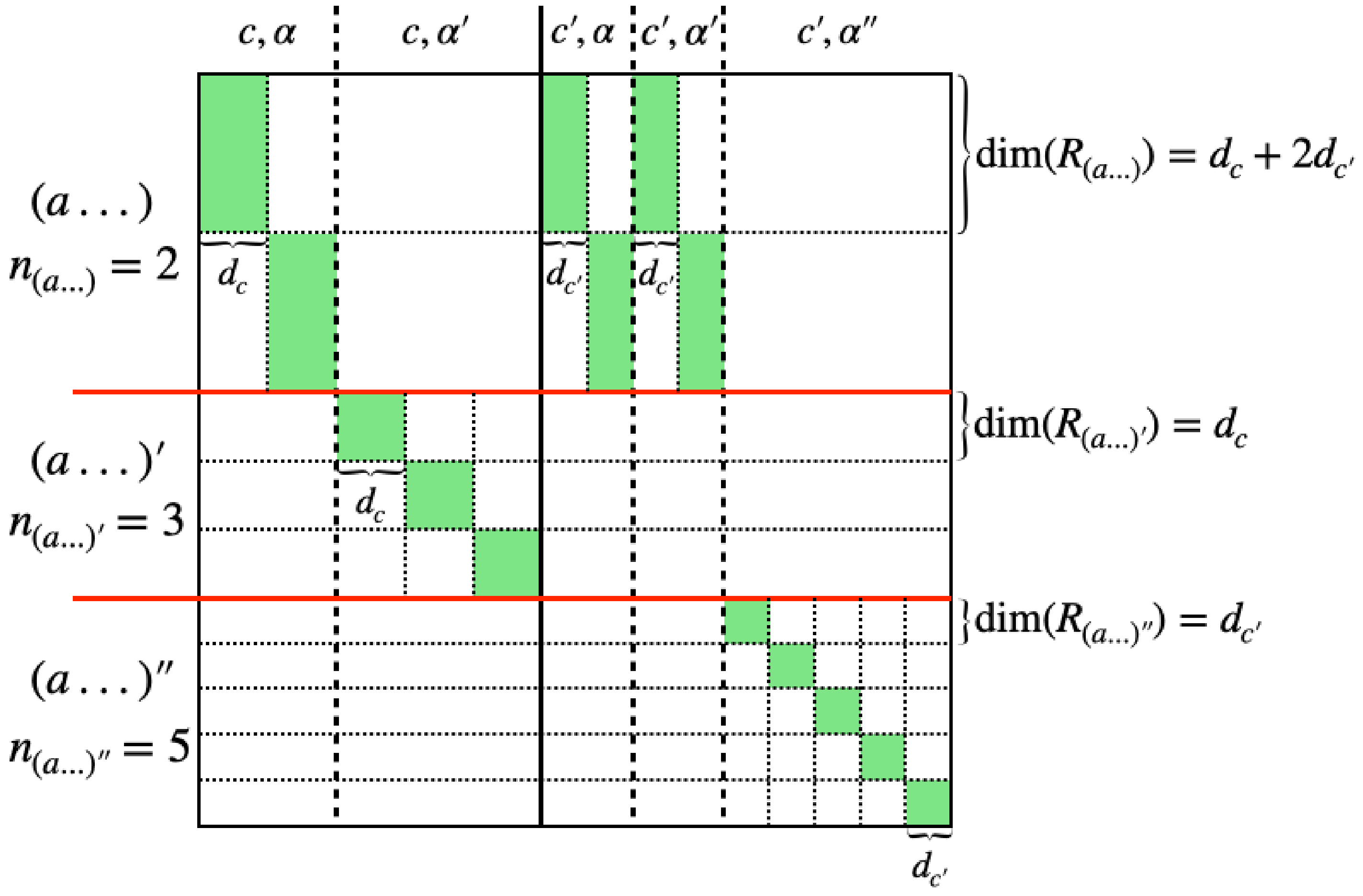


In this diagram, we have indicated how the tensor map can be rewritten in terms of a block diagonal matrix with a unitary matrix on its left and another unitary matrix (if domain and codomain are different) on its right. So the left and right matrices should actually have been drawn as squares. They represent the unitary basis transform. In this picture, the white regions are zero.

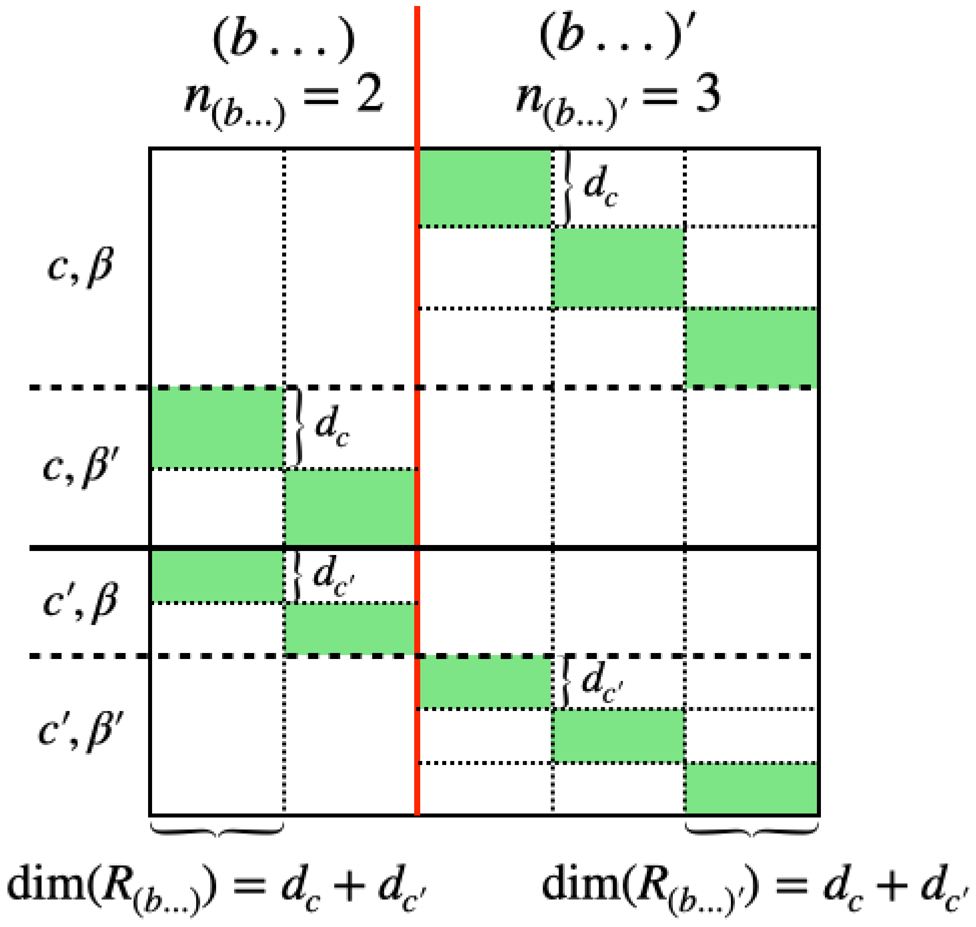


The center matrix is most easy to interpret. It is the block diagonal matrix $\bigoplus_c B_c \otimes \mathbb{1}_c$ with diagonal blocks labeled by the coupled charge `c`, in this case it takes two values. Every single small square in between the dotted or dashed lines has size $d_c \times d_c$ and corresponds to a single element of B_c, tensored with the identity $\mathbb{1}_c$. The elements of B_c are labeled by $t^c_{(a_1 \ldots a_{N_1})\alpha, (b_1 \ldots b_{N_2})\beta}$ where α labels different fusion trees from $(a_1 \ \ldots \ a_{N_1})$ to c. The dashed horizontal lines indicate regions corresponding to different fusion (actually splitting) trees, either because of different sectors $(a_1 \ \ldots \ a_{N_1})$ or different labels α within the same sector. Similarly, the dashed vertical lines define the border between regions of different fusion trees from the domain to `c`, either because of different sectors $(b_1 \ \ldots \ b_{N_2})$ or a different label β.

To understand this better, we need to understand the basis transform, e.g. on the left (codomain) side. In more detail, it is given by



Indeed, remembering that $V_i = \bigoplus_{a_i} \mathbb{C}^{n_{a_i}} \otimes R_{a_i}$ with R_{a_i} the representation space on which irrep a_i acts (with dimension $\dim(a_i)$), we find $V_1 \otimes \ldots \otimes V_{N_1} = \bigoplus_{a_1,\ldots,a_{N_1}} \mathbb{C}^{n_{a_1} \times \ldots n_{a_{N_1}}} \otimes (R_{a_1} \otimes \ldots \otimes R_{a_{N_1}})$. In the diagram above, the red lines correspond to the direct sum over the different sectors $(a_1, \ldots, a_{N_1})$, there depicted taking three possible values $(a \ldots)$, $(a \ldots)'$ and $(a \ldots)''$. The tensor product $\mathbb{C}^{n_{a_1} \times \ldots n_{a_{N_1}}} \otimes (R_{a_1} \otimes \ldots \otimes R_{a_{N_1}})$ is depicted as $(R_{a_1} \otimes \ldots \otimes R_{a_{N_1}})^{\oplus n_{a_1} \times \ldots n_{a_{N_1}}}$, i.e. as a direct sum of the spaces $R_{(a\ldots)} = (R_{a_1} \otimes \ldots \otimes R_{a_{N_1}})$ according to the dotted horizontal lines, which repeat $n_{(a\ldots)} = n_{a_1} \times \ldots n_{a_{N_1}}$ times. In this particular example, $n_{(a\ldots)} = 2$, $n_{(a\ldots)'} = 3$ and $n_{(a\ldots)''} = 5$. The thick vertical line represents the separation between the two different coupled sectors, denoted as c and c'. Dashed vertical lines represent different ways of reaching the coupled sector, corresponding to different α. In this example, the first sector $(a \ldots)$ has one fusion tree to c, labeled by c, α, and two fusion trees to c', labeled by c', α and c', α'. The second sector has only a fusion tree to c, labeled by c, α'. The third sector only has a fusion tree to c', labeld by c', α''. Finally then, because the fusion trees do not act on the spaces $\mathbb{C}^{n_{a_1} \times \ldots n_{a_{N_1}}}$, the dotted lines which represent the different $n_{(a\ldots)} = n_{a_1} \times \ldots n_{a_{N_1}}$ dimensions are also drawn vertically. In particular, for a given sector $(a \ldots)$ and a specific fusion tree $X^{(a\ldots)}_{c,\alpha} : R_c \to R_{(a\ldots)}$, the action is $\mathbb{1}_{n_{(a\ldots)}} \otimes X^{(a\ldots)}_{c,\alpha}$, which corresponds to the diagonal green blocks in this drawing where the same matrix $X^{(a\ldots)}_{c,\alpha}$ (the fusion tree) is repeated along the diagonal. Note that the fusion tree is not a vector or single column, but a matrix with number of rows equal to $\dim(R_{(a\ldots)}) = d_{a_1} d_{a_2} \ldots d_{a_{N_1}}$ and number of columns equal to d_c.



A similar interpretation can be given to the basis transform on the right, by taking its adjoint. In this particular example, it has two different combinations of sectors $(b\ \ldots)$ and $(b\ \ldots)'$, where both have a single fusion tree to c as well as to c', and $n_{(b\ldots)} = 2, n_{(b\ldots)'} = 3$.

Note that we never explicitly store or act with the basis transforms on the left and the right. For composing tensor maps (i.e. multiplying them), these basis transforms just cancel, whereas for tensor factorizations they just go through trivially. They transform non-trivially when reshuffling the tensor indices, both within or in between the domain and codomain. For this, however, we can completely rely on the manipulations of fusion trees to implicitly compute the effect of the basis transform and construct the new blocks B_c that result with respect to the new basis.

Hence, as before, we only store the diagonal blocks B_c of size `(blockdim(codomain(t), c), blockdim(domain(t), c))` as a `DenseMatrix`, accessible via `block(t, c)`. Within this matrix, there are regions of the form `view(block(t, c), m₁:m₂, n₁:n₂)` that correspond to the data $t^c_{(a_1\ldots a_{N_1})\alpha,(b_1\ldots b_{N_2})\beta}$ associated with a pair of fusion trees $X^{(a_1\ldots a_{N_1})}_{c,\alpha}$ and $X^{(b_1\ldots b_{N_2})}_{c,\beta}$, henceforth again denoted as `f₁` and `f₂`, with `f₁.coupled == f₂.coupled == c`. The ranges where this subblock is living are managed within the tensor implementation, and these subblocks can be accessed via `t[f₁,f₂]`, and is returned as a `StridedArray` of size $n_{a_1} \times n_{a_2} \times \ldots \times n_{a_{N_1}} \times n_{b_1} \times \ldots n_{b_{N_2}}$, or in code, `(dim(V1, a1), dim(V2, a2), …, dim(VN₁, aN₁), dim(W1, b1), …, dim(WN₂, bN₂))`. While the implementation does not distinguish between `FusionStyle isa UniqueFusion` or `FusionStyle isa MultipleFusion`, in the former case the fusion tree is completely characterized by the uncoupled sectors, and so the subblocks can also be accessed as `t[a1, …, aN₁,b1', …, bN₂']`. When there is no symmetry at all, i.e. `sectortype(t) == Trivial`, `t[]` returns the raw tensor data as a `StridedArray` of size `(dim(V1), …, dim(VN₁), dim(W1), …, dim(WN₂))`, whereas `block(t, Trivial())` returns the same data as a `DenseMatrix` of size `(dim(V1) * … * dim(VN₁), dim(W1) * … * dim(WN₂))`.

Constructing tensor maps and accessing tensor data

Having learned how a tensor is represented and stored, we can now discuss how to create tensors and tensor maps. From hereon, we focus purely on the interface rather than the implementation.

Random and uninitialized tensor maps

The most convenient set of constructors are those that construct tensors or tensor maps with random or uninitialized data. They take the form

```
TensorMap(f, codomain, domain)
TensorMap(f, eltype::Type{<:Number}, codomain, domain)
TensorMap(undef, codomain, domain)
TensorMap(undef, eltype::Type{<:Number}, codomain, domain)
```

Here, in the first form, `f` can be any function or object that is called with an argument of type `Dims{2} = Tuple{Int,Int}` and is such that `f((m,n))` creates a `DenseMatrix` instance with `size(f(m,n)) == (m,n)`. In the second form, `f` is called as `f(eltype,(m,n))`. Possibilities for `f` are `randn` and `rand` from Julia Base. TensorXD.jl provides `randnormal` and `randuniform` as an synonym for `randn` and `rand`, as well as the new function `randisometry`, alternatively called `randhaar`, that creates a random isometric `m × n` matrix `w` satisfying `w'*w ≈ I` distributed according to the Haar measure (this requires `m>= n`). The third and fourth calling syntax use the `UndefInitializer` from Julia Base and generates a `TensorMap` with unitialized data, which could thus contain `NaN`s.

In all of these constructors, the last two arguments can be replaced by `domain→codomain` or `codomain←domain`, where the arrows are obtained as `\rightarrow+TAB` and `\leftarrow+TAB` and create a `HomSpace` as explained in the section on Spaces of morphisms. Some examples are perhaps in order

```
julia> t1 = TensorMap(randnormal, ℂ^2 ⊗ ℂ^3, ℂ^2)
TensorMap((ℂ^2 ⊗ ℂ^3) ← ProductSpace(ℂ^2)):
[:, :, 1] =
 -0.6728102491657165   -0.40445997866617917  -0.3995302664733455
 -0.10157547019874053  -0.1926110497446887    0.34551648737850466

[:, :, 2] =
 3.0961147940500373   -0.4176970655809272    0.13121093439860804
 3.0961147940360373  -0.41709760355809272   0.13121093439800804
 1.143596716489897   -0.6908049537738421  -1.3837680507332004

julia> t2 = TensorMap(randisometry, Float32, ℂ^2 ⊗ ℂ^3 ← ℂ^2)
TensorMap((ℂ^2 ⊗ ℂ^3) ← ProductSpace(ℂ^2)):
[:, :, 1] =
  0.23204255f0  -0.26526508f0  -0.3162529f0
 -0.19685945f0  -0.84341264f0   0.16023846f0

[:, :, 2] =
 -0.43310603f0   0.08323157f0   0.19591664f0
 -0.8210975f0   -0.08287482f0  -0.293324f0

julia> t3 = TensorMap(undef, ℂ^2 → ℂ^2 ⊗ ℂ^3)
TensorMap((ℂ^2 ⊗ ℂ^3) ← ProductSpace(ℂ^2)):
[:, :, 1] =
 6.9134757270492e-310   6.91345438057146e-310  6.9134543805778e-310
 6.91345438056987e-310  6.9134543805746e-310   6.91345438057936e-310

[:, :, 2] =
 6.9134543805825e-310  6.9134543805857e-310   6.91345438058885e-310
 6.9134543805841e-310  6.91345438058727e-310  6.91345524605115e-310

julia> domain(t1) == domain(t2) == domain(t3)
true

julia> codomain(t1) == codomain(t2) == codomain(t3)
true

julia> disp(x) = show(IOContext(Core.stdout, :compact=>false), "text/plain", trunc.(x;

julia> t1[] |> disp
2×3×2 Strided.StridedView{Float64,3,Array{Float64,1},typeof(identity)}:
[:, :, 1] =
 -0.672  -0.404  -0.399
 -0.101  -0.192   0.345

[:, :, 2] =
 3.096  -0.417   0.131
 1.143  -0.69   -1.383
julia> block(t1, Trivial()) |> disp
6×2 Array{Float64,2}:
 -0.672   3.096
 -0.101   1.143
 -0.404  -0.417
 -0.192  -0.69

 -0.399   0.131
 -0.399   0.131
  0.345  -1.383
julia> reshape(t1[], dim(codomain(t1)), dim(domain(t1))) |> disp
6×2 Array{Float64,2}:
 -0.672   3.096
 -0.101   1.143
 -0.404  -0.417
 -0.192  -0.69
 -0.399   0.131
  0.345  -1.383
```

Finally, all constructors can also be replaced by `Tensor(..., codomain)`, in which case the domain is assumed to be the empty `ProductSpace{S,0}()`, which can easily be obtained as `one(codomain)`. Indeed, the empty product space is the unit object of the monoidal category, equivalent to the field of scalars 𝕜, and thus the multiplicative identity (especially since `*` also acts as tensor product on vector spaces).

The matrices created by `f` are the matrices B_c discussed above, i.e. those returned by `block(t, c)`. Only numerical matrices of type `DenseMatrix` are accepted, which in practice just means Julia's intrinsic `Matrix{T}` for some `T<:Number`. In the future, we will add support for `CuMatrix` from CuArrays.jl to harness GPU computing power, and maybe `SharedArray` from the Julia's `SharedArrays` standard library.

Support for static or sparse data is currently unavailable, and if it would be implemented, it would lead to new subtypes of `AbstractTensorMap` which are distinct from `TensorMap`. Future implementations of e.g. `SparseTensorMap` or `StaticTensorMap` could be useful. Furthermore, there could be specific implementations for tensors whose blocks are `Diagonal`.

Tensor maps from existing data

To create a `TensorMap` with existing data, one can use the aforementioned form but with the function `f` replaced with the actual data, i.e. `TensorMap(data, codomain, domain)` or any of its equivalents.

Here, `data` can be of two types. It can be a dictionary (any `Associative` subtype) which has blocksectors `c` of type `sectortype(codomain)` as keys, and the corresponding matrix blocks as value, i.e. `data[c]` is some `DenseMatrix` of size `(blockdim(codomain, c), blockdim(domain, c))`. This is the form of how the data is stored within the `TensorMap` objects.

For those space types for which a `TensorMap` can be converted to a plain multidimensional array, the `data` can also be a general `DenseArray`, either of rank `N₁+N₂` and with matching size `(dims(codomain)..., dims(domain)...)`, or just as a `DenseMatrix` with size `(dim(codomain), dim(domain))`. This is true in particular if the sector type is `Trivial`, e.g. for `CartesianSpace` or `ComplexSpace`. Then the `data` array is just reshaped into matrix form and referred to as such in the resulting `TensorMap` instance. When `spacetype` is `GradedSpace`, the `TensorMap` constructor will try to reconstruct the tensor data such that the resulting tensor `t` satisfies `data == convert(Array, t)`. This might not be possible, if the data does not respect the symmetry structure. Let's sketch this with a simple example

```
julia> data = zeros(2,2,2,2)
2×2×2×2 Array{Float64,4}:
[:, :, 1, 1] =
 0.0  0.0
 0.0  0.0

[:, :, 2, 1] =
 0.0  0.0
 0.0  0.0

[:, :, 1, 2] =
 0.0  0.0
 0.0  0.0

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

julia> # encode the operator (σ_x * σ_x + σ_y * σ_y + σ_z * σ_z)/4
       # that is, the swap gate, which maps the last two indices on the first two in r
       # also known as Heisenberg interaction between two spin 1/2 particles
       data[1,2,2,1] = data[2,1,1,2] = 1/2
0.5

julia> data[1,1,1,1] = data[2,2,2,2] = 1/4
0.25

julia> data[1,2,1,2] = data[2,1,2,1] = -1/4
-0.25

julia> V1 = ℂ^2 # generic qubit hilbert space
ℂ^2

julia> t1 = TensorMap(data, V1 ⊗ V1, V1 ⊗ V1)
TensorMap((ℂ^2 ⊗ ℂ^2) ← (ℂ^2 ⊗ ℂ^2)):
[:, :, 1, 1] =
 0.25  0.0
 0.0   0.0

[:, :, 2, 1] =
  0.0   0.5
 -0.25  0.0
```

```
[:, :, 1, 2] =
 0.0  -0.25
 0.5   0.0

[:, :, 2, 2] =
 0.0  0.0
 0.0  0.25

julia> V2 = SU2Space(1/2=>1) # hilbert space of an actual spin-1/2 particle, respectin
Rep[SU₂](1/2=>1)

julia> t2 = TensorMap(data, V2 ⊗ V2, V2 ⊗ V2)
TensorMap((Rep[SU₂](1/2=>1) ⊗ Rep[SU₂](1/2=>1)) ← (Rep[SU₂](1/2=>1) ⊗ Rep[SU₂](1/2=>1)
* Data for fusiontree FusionTree{Irrep[SU₂]}((1/2, 1/2), 0, (false, false), ()) ← Fusi
[:, :, 1, 1] =
 -0.7500000000000002
* Data for fusiontree FusionTree{Irrep[SU₂]}((1/2, 1/2), 1, (false, false), ()) ← Fusi
[:, :, 1, 1] =
 0.25

julia> V3 = U1Space(1/2=>1,-1/2=>1) # restricted space that only uses the `σ_z` rotati
Rep[U₁](1/2=>1, -1/2=>1)

julia> t3 = TensorMap(data, V3 ⊗ V3, V3 ⊗ V3)
TensorMap((Rep[U₁](1/2=>1, -1/2=>1) ⊗ Rep[U₁](1/2=>1, -1/2=>1)) ← (Rep[U₁](1/2=>1, -1/
* Data for sector (Irrep[U₁](-1/2), Irrep[U₁](1/2)) ← (Irrep[U₁](-1/2), Irrep[U₁](1/2)
[:, :, 1, 1] =
 -0.25
* Data for sector (Irrep[U₁](1/2), Irrep[U₁](-1/2)) ← (Irrep[U₁](-1/2), Irrep[U₁](1/2)
[:, :, 1, 1] =
 0.5
* Data for sector (Irrep[U₁](-1/2), Irrep[U₁](1/2)) ← (Irrep[U₁](1/2), Irrep[U₁](-1/2)
[:, :, 1, 1] =
 0.5
* Data for sector (Irrep[U₁](1/2), Irrep[U₁](-1/2)) ← (Irrep[U₁](1/2), Irrep[U₁](-1/2)
[:, :, 1, 1] =
 -0.25
* Data for sector (Irrep[U₁](1/2), Irrep[U₁](1/2)) ← (Irrep[U₁](1/2), Irrep[U₁](1/2)):
[:, :, 1, 1] =
 0.25
* Data for sector (Irrep[U₁](-1/2), Irrep[U₁](-1/2)) ← (Irrep[U₁](-1/2), Irrep[U₁](-1/
[:, :, 1, 1] =
 0.25

julia> for (c,b) in blocks(t3)
           println("Data for block $c :")
           println("Data for block $c :")
           b |> disp
           println()
       end
Data for block Irrep[U₁](0) :
2×2 Array{Float64,2}:
 -0.25   0.5
  0.5   -0.25
Data for block Irrep[U₁](1) :
1×1 Array{Float64,2}:
 0.25
Data for block Irrep[U₁](-1) :
1×1 Array{Float64,2}:
 0.25
```

Hence, we recognize that the Heisenberg interaction has eigenvalue $-3/4$ in the coupled spin zero sector (`SUIrrep(0)`), and eigenvalue $+1/4$ in the coupled spin 1 sector (`SU2Irrep(1)`). Using `Irrep[U₁]` instead, we observe that both coupled charge `U1Irrep(+1)` and `U1Irrep(-1)` have eigenvalue $+1/4$. The coupled charge `U1Irrep(0)` sector is two-dimensional, and has an eigenvalue $+1/4$ and an eigenvalue $-3/4$.

To construct the proper `data` in more complicated cases, one has to know where to find each sector in the range `1:dim(V)` of every index `i` with associated space `V`, as well as the internal structure of the representation space when the corresponding sector `c` has `dim(c)>1`, i.e. in the case of `FusionStyle(c) isa MultipleFusion`. Currently, the only non- abelian sectors are `Irrep[SU₂]` and `Irrep[CU₁]`, for which the internal structure is the natural one.

There are some tools available to facilate finding the proper range of sector `c` in space `V`, namely `axes(V, c)`. This also works on a `ProductSpace`, with a tuple of sectors. An example

```
julia> V = SU2Space(0=>3, 1=>2, 2=>1)
Rep[SU₂](0=>3, 1=>2, 2=>1)

julia> P = V ⊗ V ⊗ V
(Rep[SU₂](0=>3, 1=>2, 2=>1) ⊗ Rep[SU₂](0=>3, 1=>2, 2=>1) ⊗ Rep[SU₂](0=>3, 1=>2, 2=>1))

julia> axes(P, (SU2Irrep(1), SU2Irrep(0), SU2Irrep(2)))
(4:9, 1:3, 10:14)
```

Note that the length of the range is the degeneracy dimension of that sector, times the dimension of the internal representation space, i.e. the quantum dimension of that sector.

Constructing similar tensors

A third way to construct a `TensorMap` instance is to use `Base.similar`, i.e.

```
similar(t [, T::Type{<:Number}, codomain, domain])
```

where `T` is a possibly different `eltype` for the tensor data, and `codomain` and `domain` optionally define a new codomain and domain for the resulting tensor. By default, these values just take the value from the input tensor `t`. The result will be a new `TensorMap` instance, with `undef` data, but whose data is stored in the same subtype of `DenseMatrix` (e.g. `Matrix` or `CuMatrix` or ...) as `t`. In particular, this uses the methods `storagetype(t)` and `TensorXD.similarstoragetype(t, T)`.

Special purpose constructors

Finally, there are methods `zero`, `one`, `id`, `isomorphism`, `unitary` and `isometry` to create specific new tensors. Tensor maps behave as vectors and can be added (if they have the same domain and codomain); `zero(t)` is the additive identity, i.e. a `TensorMap` instance where all entries are zero. For a `t::TensorMap` with `domain(t) == codomain(t)`, i.e. an endomorphism, `one(t)` creates the identity tensor, i.e. the identity under composition. As discussed in the section on linear algebra operations, we denote composition of tensor maps with the mutliplication operator `*`, such that `one(t)` is the multiplicative identity. Similarly, it can be created as `id(V)` with `V` the relevant vector space, e.g. `one(t) == id(domain(t))`. The identity tensor is currently represented with dense data, and one can use `id(A::Type{<:DenseMatrix}, V)` to specify the type of `DenseMatrix` (and its `eltype`), e.g. `A = Matrix{Float64}`. Finally, it often occurs that we want to construct a specific isomorphism between two spaces that are isomorphic but not equal, and for which there is no canonical choice. Hereto, one can use the method `u = isomorphism([A::Type{<:DenseMatrix}, ] codomain, domain)`, which will explicitly check that the domain and codomain are isomorphic, and return an error otherwise. Again, an optional first argument can be given to specify the specific type of `DenseMatrix` that is currently used to store the rather trivial data of this tensor. If `spacetype(u) <: EuclideanSpace`, the same result can be obtained with the method `u = unitary([A::Type{<:DenseMatrix}, ] codomain, domain)`. Note that reversing the domain and codomain yields the inverse morphism, which in the case of `EuclideanSpace` coincides with the adjoint morphism, i.e. `isomorphism(A, domain, codomain) == adjoint(u) == inv(u)`, where `inv` and `adjoint` will be further discussed below. Finally, if two spaces `V1` and `V2` are such that `V2` can be embedded in `V1`, i.e. there exists an inclusion with a left inverse, and furthermore they represent tensor products of some `EuclideanSpace`, the function `w = isometry([A::Type{<:DenseMatrix}, ], V1, V2)` creates one specific isometric embedding, such that `adjoint(w)*w == id(V2)` and `w*adjoint(w)` is some hermitian idempotent (a.k.a. orthogonal projector) acting on `V1`. An error will be thrown if such a map cannot be constructed for the given domain and codomain.

Let's conclude this section with some examples with `GradedSpace`.

```
julia> V1 = ℤ₂Space(0=>3,1=>2)
Rep[ℤ₂](0=>3, 1=>2)
```

```
Rep[ℤ₂](0=>2, 1=>1)

julia> # First a `TensorMap{ℤ₂Space, 1, 1}`
       m = TensorMap(randn, V1, V2)
TensorMap(ProductSpace(Rep[ℤ₂](0=>3, 1=>2)) ← ProductSpace(Rep[ℤ₂](0=>2, 1=>1))):
* Data for sector (Irrep[ℤ₂](0),) ← (Irrep[ℤ₂](0),):
 1.2633100061037716  -0.3427795634064988
 1.0329531127199874  -0.16005168123568242
 1.240098385185237   -0.3843443751356317
* Data for sector (Irrep[ℤ₂](1),) ← (Irrep[ℤ₂](1),):
  2.20596692446051
 -0.4887404085551539

julia> convert(Array, m) |> disp
5×3 Array{Float64,2}:
 1.263  -0.342   0.0
 1.032  -0.16    0.0
 1.24   -0.384   0.0
 0.0     0.0     2.205
 0.0     0.0    -0.488
julia> # compare with:
       block(m, Irrep[ℤ₂](0)) |> disp
3×2 Array{Float64,2}:
 1.263  -0.342
 1.032  -0.16
 1.24   -0.384
julia> block(m, Irrep[ℤ₂](1)) |> disp
2×1 Array{Float64,2}:
  2.205
 -0.488
julia> # Now a `TensorMap{ℤ₂Space, 2, 2}`
       t = TensorMap(randn, V1 ⊗ V1, V2 ⊗ V2')
TensorMap((Rep[ℤ₂](0=>3, 1=>2) ⊗ Rep[ℤ₂](0=>3, 1=>2)) ← (Rep[ℤ₂](0=>2, 1=>1) ⊗ Rep[ℤ₂]
* Data for sector (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](0)):
[:, :, 1, 1] =
 -0.03513598153957989   1.4648464324395989
  0.9632835530565921   -1.6026999068122585

[:, :, 2, 1] =
 -0.9606127063113135   0.6752596209659316
  0.8977772575364383  -0.8159968592614363

[:, :, 1, 2] =
 -1.0937006025272407   -0.0032621948016960485
  0.22068646038022152  -1.068738406438155
```

```
[:, :, 2, 2] =
  0.6541789428021042  -1.2409289509741461
 -1.525883365440826    0.12399970560716514
* Data for sector (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](0)):
[:, :, 1, 1] =
 -0.7234151165051018   -1.2902220438135208  -0.4565382989223469
  0.02120986575730163  -0.1414342568674333  -0.8409824997787858
 -0.7001208370367475   -0.4449778930551481   0.5809958442755666

[:, :, 2, 1] =
 0.6049976663428605   -1.4259476040999772   -0.8253325775592087
 0.47639610049457454   0.21801971355103067   0.061813647219221154
 0.3897523389585391   -0.38668734722490156   0.6409067529460759

[:, :, 1, 2] =
 -0.8430309269198787  0.7045804761125098   -1.1589170562431441
 -0.6762092724797707  0.6584873076703133    0.2880477828780876
 -0.8279599428420612  0.14130962250012002   0.5504368639003666

[:, :, 2, 2] =
 -1.1143341768389161  0.09080613837043017  -0.42209787207876337
  1.8790119106272714  1.5217720783697788    1.6675907722910381
 -0.6553994381398371  1.8322330836336043   -0.8009479911861984
* Data for sector (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](1)):
[:, :, 1, 1] =
 -1.1700709270920926  -0.8368429318818991
 -0.9368303298268121  -2.0262279751859706
* Data for sector (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](1)):
[:, :, 1, 1] =
  0.13560773884572913   0.818132019096569   1.009297255176382
 -0.5083565312867032    0.6232255733393918  1.6583082762923278
 -0.0317841956130747   -0.6311452002221776  0.685381518023461
* Data for sector (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](0)):
[:, :, 1, 1] =
  0.4469690010631125  -0.4797719459317822  -0.13641103168873803
 -1.697094018650829    1.79429669248575     0.638853913175749

[:, :, 1, 2] =
  1.45029109961941    0.12764896882438798  0.22544896441395015
 -1.0017874263849642  1.221481903811265    0.7357179002737811
* Data for sector (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](0)):
[:, :, 1, 1] =
  0.18777442285173643    0.7589190260703597
  0.029100339389128645  -0.29042735203766734
 -1.7668606097633786    -1.033110220138864
```

```
[:, :, 1, 2] =
 -1.0791938829626457  -0.9110586201024055
 -1.2727283829729652  -0.1559818514479427
 -1.0993207523060966   0.9240432736884797
* Data for sector (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](1)):
[:, :, 1, 1] =
 0.2156819046897459   -1.3091008197507146   -1.844358708204597
 0.34991004966887246  -0.35526703714261326  -1.856916932684818

[:, :, 2, 1] =
  0.3461643941472437   -1.2059252305529624    0.8034968836379958
 -0.28187016080815436   0.008604040590898913  0.08391693071166295
* Data for sector (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](1)):
[:, :, 1, 1] =
  0.18569194318275153   1.5852991344586238
  1.180019532326l162   -0.11193089107675376
 -0.7530356075866116    1.086967088373101

[:, :, 2, 1] =
 -0.9444431330841963    -0.8438056477561465
  0.033212995215905664  -1.0851299771124534
  0.18663725876232803   -1.3282276068410446

julia> (array = convert(Array, t)) |> disp
5×5×3×3 Array{Float64,4}:
[:, :, 1, 1] =
 -0.723  -1.29   -0.456   0.0     0.0
  0.021  -0.141  -0.84    0.0     0.0
 -0.7    -0.444   0.58    0.0     0.0
  0.0     0.0     0.0    -0.035   1.464
  0.0     0.0     0.0     0.963  -1.602

[:, :, 2, 1] =
 0.604  -1.425  -0.825   0.0     0.0
 0.476   0.218   0.061   0.0     0.0
 0.389  -0.386   0.64    0.0     0.0
 0.0     0.0     0.0    -0.96    0.675
 0.0     0.0     0.0     0.897  -0.815

[:, :, 3, 1] =
  0.0     0.0     0.0     0.187   0.758
  0.0     0.0     0.0     0.029  -0.29
  0.0     0.0     0.0    -1.766  -1.033
  0.446  -0.479  -0.136   0.0     0.0
 -1.697   1.794   0.638   0.0     0.0
```

```
[:, :, 1, 2] =
 -0.843  0.704  -1.158   0.0     0.0
 -0.676  0.658   0.288   0.0     0.0
 -0.827  0.141   0.55    0.0     0.0
  0.0    0.0     0.0    -1.093  -0.003
  0.0    0.0     0.0     0.22   -1.068

[:, :, 2, 2] =
 -1.114  0.09   -0.422   0.0     0.0
  1.879  1.521   1.667   0.0     0.0
 -0.655  1.832  -0.8     0.0     0.0
  0.0    0.0     0.0     0.654  -1.24
  0.0    0.0     0.0    -1.525   0.123

[:, :, 3, 2] =
  0.0    0.0    0.0    -1.079  -0.911
  0.0    0.0    0.0    -1.272  -0.155
  0.0    0.0    0.0    -1.099   0.924
  1.45   0.127  0.225   0.0     0.0
 -1.001  1.221  0.735   0.0     0.0

[:, :, 1, 3] =
 0.0     0.0     0.0     0.185   1.585
 0.0     0.0     0.0     1.18   -0.111
 0.0     0.0     0.0    -0.753   1.086
 0.215  -1.309  -1.844   0.0     0.0
 0.349  -0.355  -1.856   0.0     0.0

[:, :, 2, 3] =
  0.0     0.0    0.0    -0.944  -0.843
  0.0     0.0    0.0     0.033  -1.085
  0.0     0.0    0.0     0.186  -1.328
  0.346  -1.205  0.803   0.0     0.0
 -0.281   0.008  0.083   0.0     0.0

[:, :, 3, 3] =
  0.135   0.818  1.009   0.0     0.0
 -0.508   0.623  1.658   0.0     0.0
 -0.031  -0.631  0.685   0.0     0.0
  0.0     0.0    0.0    -1.17   -0.836
  0.0     0.0    0.0    -0.936  -2.026
julia> d1 = dim(codomain(t))
25

julia> d2 = dim(domain(t))

9
```

```
9

julia> (matrix = reshape(array, d1, d2)) |> disp
25×9 Array{Float64,2}:
 -0.723   0.604   0.0    -0.843  -1.114   0.0     0.0     0.0     0.135
  0.021   0.476   0.0    -0.676   1.879   0.0     0.0     0.0    -0.508
 -0.7     0.389   0.0    -0.827  -0.655   0.0     0.0     0.0    -0.031
  0.0     0.0     0.446   0.0     0.0     1.45    0.215   0.346   0.0
  0.0     0.0    -1.697   0.0     0.0    -1.001   0.349  -0.281   0.0
 -1.29   -1.425   0.0     0.704   0.09    0.0     0.0     0.0     0.818
 -0.141   0.218   0.0     0.658   1.521   0.0     0.0     0.0     0.623
 -0.444  -0.386   0.0     0.141   1.832   0.0     0.0     0.0    -0.631
  0.0     0.0    -0.479   0.0     0.0     0.127  -1.309  -1.205   0.0
  0.0     0.0     1.794   0.0     0.0     1.221  -0.355   0.008   0.0
 -0.456  -0.825   0.0    -1.158  -0.422   0.0     0.0     0.0     1.009
 -0.84    0.061   0.0     0.288   1.667   0.0     0.0     0.0     1.658
  0.58    0.64    0.0     0.55   -0.8     0.0     0.0     0.0     0.685
  0.0     0.0    -0.136   0.0     0.0     0.225  -1.844   0.803   0.0
  0.0     0.0     0.638   0.0     0.0     0.735  -1.856   0.083   0.0
  0.0     0.0     0.187   0.0     0.0    -1.079   0.185  -0.944   0.0
  0.0     0.0     0.029   0.0     0.0    -1.272   1.18    0.033   0.0
  0.0     0.0    -1.766   0.0     0.0    -1.099  -0.753   0.186   0.0
 -0.035  -0.96    0.0    -1.093   0.654   0.0     0.0     0.0    -1.17
  0.963   0.897   0.0     0.22   -1.525   0.0     0.0     0.0    -0.936
  0.0     0.0     0.758   0.0     0.0    -0.911   1.585  -0.843   0.0
  0.0     0.0    -0.29    0.0     0.0    -0.155  -0.111  -1.085   0.0
  0.0     0.0    -1.033   0.0     0.0     0.924   1.086  -1.328   0.0
  1.464   0.675   0.0    -0.003  -1.24    0.0     0.0     0.0    -0.836
 -1.602  -0.815   0.0    -1.068   0.123   0.0     0.0     0.0    -2.026
julia> (u = reshape(convert(Array, unitary(codomain(t), fuse(codomain(t)))), d1, d1))
25×25 Array{Float64,2}:
 1.0  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  1.0  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  1.0  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  1.0  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  1.0  0.0  0.0
 0.0  0.0  0.0  1.0  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  1.0  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  1.0  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  1.0  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  1.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  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  1.0  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  1.0  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  0.0  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  0.0  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  0.0  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  1.0  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  1.0  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  0.0  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  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  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  1.0  0.0  0.0  0.0  0.0
julia> (v = reshape(convert(Array, unitary(domain(t), fuse(domain(t)))), d2, d2)) |> d
9×9 Array{Float64,2}:
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
julia> u'*u ≈ I ≈ v'*v
true

julia> (u'*matrix*v) |> disp
25×9 Array{Float64,2}:
 -0.723   0.604  -0.843  -1.114   0.135   0.0     0.0     0.0     0.0
  0.021   0.476  -0.676   1.879  -0.508   0.0     0.0     0.0     0.0
 -0.7     0.389  -0.827  -0.655  -0.031   0.0     0.0     0.0     0.0
 -1.29   -1.425   0.704   0.09    0.818   0.0     0.0     0.0     0.0
 -0.141   0.218   0.658   1.521   0.623   0.0     0.0     0.0     0.0
 -0.444  -0.386   0.141   1.832  -0.631   0.0     0.0     0.0     0.0
 -0.456  -0.825  -1.158  -0.422   1.009   0.0     0.0     0.0     0.0
 -0.84    0.061   0.288   1.667   1.658   0.0     0.0     0.0     0.0
  0.58    0.64    0.55   -0.8     0.685   0.0     0.0     0.0     0.0
 -0.035  -0.96   -1.093   0.654  -1.17    0.0     0.0     0.0     0.0
  0.963   0.897   0.22   -1.525  -0.936   0.0     0.0     0.0     0.0
  1.464   0.675  -0.003  -1.24   -0.836   0.0     0.0     0.0     0.0
 -1.602  -0.815  -1.068   0.123  -2.026   0.0     0.0     0.0     0.0
  0.0     0.0     0.0     0.0     0.0     0.446   1.45    0.215   0.346
  0.0     0.0     0.0     0.0     0.0    -1.697  -1.001   0.349  -0.281
  0.0     0.0     0.0     0.0     0.0    -0.479   0.127  -1.309  -1.205
  0.0     0.0     0.0     0.0     0.0     1.794   1.221  -0.355   0.008
  0.0     0.0     0.0     0.0     0.0    -0.136   0.225  -1.844   0.803
  0.0     0.0     0.0     0.0     0.0     0.638   0.735  -1.856   0.083
  0.0     0.0     0.0     0.0     0.0     0.187  -1.079   0.185  -0.944

  0.0     0.0     0.0     0.0     0.0     0.029  -1.272   1.18    0.033
  0.0     0.0     0.0     0.0     0.0     0.029  -1.272   1.18    0.033
  0.0     0.0     0.0     0.0     0.0    -1.766  -1.099  -0.753   0.186
  0.0     0.0     0.0     0.0     0.0     0.758  -0.911   1.585  -0.843
  0.0     0.0     0.0     0.0     0.0    -0.29   -0.155  -0.111  -1.085
  0.0     0.0     0.0     0.0     0.0    -1.033   0.924   1.086  -1.328
julia> # compare with:
       block(t, Z2Irrep(0)) |> disp
13×5 Array{Float64,2}:
 -0.723   0.604  -0.843  -1.114   0.135
  0.021   0.476  -0.676   1.879  -0.508
 -0.7     0.389  -0.827  -0.655  -0.031
 -1.29   -1.425   0.704   0.09    0.818
 -0.141   0.218   0.658   1.521   0.623
 -0.444  -0.386   0.141   1.832  -0.631
 -0.456  -0.825  -1.158  -0.422   1.009
 -0.84    0.061   0.288   1.667   1.658
  0.58    0.64    0.55   -0.8     0.685
 -0.035  -0.96   -1.093   0.654  -1.17
  0.963   0.897   0.22   -1.525  -0.936
  1.464   0.675  -0.003  -1.24   -0.836
 -1.602  -0.815  -1.068   0.123  -2.026
julia> block(t, Z2Irrep(1)) |> disp
12×4 Array{Float64,2}:
  0.446   1.45    0.215   0.346
 -1.697  -1.001   0.349  -0.281
 -0.479   0.127  -1.309  -1.205
  1.794   1.221  -0.355   0.008
 -0.136   0.225  -1.844   0.803
  0.638   0.735  -1.856   0.083
  0.187  -1.079   0.185  -0.944
  0.029  -1.272   1.18    0.033
 -1.766  -1.099  -0.753   0.186
  0.758  -0.911   1.585  -0.843
 -0.29   -0.155  -0.111  -1.085
 -1.033   0.924   1.086  -1.328
```

Here, we illustrated some additional concepts. Firstly, note that we convert a `TensorMap` to an `Array`. This only works when `sectortype(t)` supports `fusiontensor`, and in particular when `BraidingStyle(sectortype(t)) == Bosonic()`, e.g. the case of trivial tensors (the category $\mathbf{Vect}$) and group representations (the category $\mathbf{Rep}_{\mathsf{G}}$, which can be interpreted as a subcategory of $\mathbf{Vect}$). Here, we are in this case with $\mathsf{G} = \mathbb{Z}_2$. For a `TensorMap{S,1,1}`, the blocks directly correspond to the diagonal blocks in the block diagonal structure of its representation as an `Array`, there is no basis transform in between. This is no longer the case for `TensorMap{S,N₁,N₂}` with different values of `N₁` and `N₂`. Here, we use the operation `fuse(V)`, which creates an `ElementarySpace` which is isomorphic to a given space `V` (of type `ProductSpace` or `ElementarySpace`). The specific map between those two spaces constructed using the specific method `unitary` implements precisely the basis change from the product basis to the coupled basis. In this case, for a group `G` with `FusionStyle(Irrep[G]) isa UniqueFusion`, it is a permutation matrix. Specifically choosing `V` equal to the codomain and domain of `t`, we can construct the explicit basis transforms that bring `t` into block diagonal form.

Let's repeat the same exercise for `I = Irrep[SU₂]`, which has `FusionStyle(I) isa MultipleFusion`.

```
julia> V1 = SU₂Space(0=>2,1=>1)
Rep[SU₂](0=>2, 1=>1)

julia> V2 = SU₂Space(0=>1,1=>1)
Rep[SU₂](0=>1, 1=>1)

julia> # First a `TensorMap{SU₂Space, 1, 1}`
       m = TensorMap(randn, V1, V2)
TensorMap(ProductSpace(Rep[SU₂](0=>2, 1=>1)) ← ProductSpace(Rep[SU₂](0=>1, 1=>1))):
* Data for fusiontree FusionTree{Irrep[SU₂]}((0,), 0, (false,), ()) ← FusionTree{Irrep
 -0.6563923539285431
  3.2090440359216394
* Data for fusiontree FusionTree{Irrep[SU₂]}((1,), 1, (false,), ()) ← FusionTree{Irrep
 -0.03366177225273526

julia> convert(Array, m) |> disp
5×4 Array{Float64,2}:
 -0.656   0.0     0.0     0.0
  3.209   0.0     0.0     0.0
  0.0    -0.033   0.0     0.0
  0.0     0.0    -0.033   0.0
  0.0     0.0     0.0    -0.033
julia> # compare with:
       block(m, Irrep[SU₂](0)) |> disp
2×1 Array{Float64,2}:
 -0.656
  3.209
julia> block(m, Irrep[SU₂](1)) |> disp
1×1 Array{Float64,2}:
 -0.033
julia> # Now a `TensorMap{SU₂Space, 2, 2}`
       t = TensorMap(randn, V1 ⊗ V1, V2 ⊗ V2')
TensorMap((Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>2, 1=>1)) ← (Rep[SU₂](0=>1, 1=>1) ⊗ Rep[
* Data for fusiontree FusionTree{Irrep[SU₂]}((1, 1), 0, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 0.6785089624817361
* Data for fusiontree FusionTree{Irrep[SU₂]}((0, 0), 0, (false, false), ()) ← FusionTr
[:, :, 1, 1] =

 -0.3780309334951461  -1.1427612599059842
 -0.3786309334951401  -1.1427612599659842
  0.7042037575692931  -0.11897792418949947
* Data for fusiontree FusionTree{Irrep[SU₂]}((1, 1), 0, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 0.985503601135925
* Data for fusiontree FusionTree{Irrep[SU₂]}((0, 0), 0, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 -0.5220840153589902   0.9608955437332395
  0.31677731075765536  0.7143157721284216
* Data for fusiontree FusionTree{Irrep[SU₂]}((1, 0), 1, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 -1.0498813732087382  0.8782560738051121
* Data for fusiontree FusionTree{Irrep[SU₂]}((0, 1), 1, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 -0.13694848200035717
 -0.08061312848832627
* Data for fusiontree FusionTree{Irrep[SU₂]}((1, 1), 1, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 3.284509238423052
* Data for fusiontree FusionTree{Irrep[SU₂]}((1, 0), 1, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 -1.1067821961065423  0.16747685441715093
* Data for fusiontree FusionTree{Irrep[SU₂]}((0, 1), 1, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 0.5433798392357461
 0.23845363134950934
* Data for fusiontree FusionTree{Irrep[SU₂]}((1, 1), 1, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 0.2067386888125478
* Data for fusiontree FusionTree{Irrep[SU₂]}((1, 0), 1, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 -0.36281622042580014  -1.3342204731693237
* Data for fusiontree FusionTree{Irrep[SU₂]}((0, 1), 1, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 0.24575412600654148
 0.8320291925128814
* Data for fusiontree FusionTree{Irrep[SU₂]}((1, 1), 1, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 1.535140001377151
* Data for fusiontree FusionTree{Irrep[SU₂]}((1, 1), 2, (false, false), ()) ← FusionTr
[:, :, 1, 1] =
 -0.24632569838495164

julia> (array = convert(Array, t)) |> disp
5×5×4×4 Array{Float64,4}:
[:, :, 1, 1] =

 -0.378  -1.142  0.0      0.0     0.0
 -0.378  -1.142  0.0     0.0    0.0
  0.704  -0.118  0.0     0.0    0.0
  0.0     0.0    0.0     0.0    0.391
  0.0     0.0    0.0    -0.391  0.0
  0.0     0.0    0.391   0.0    0.0

[:, :, 2, 1] =
  0.0    0.0    -0.136  0.0    0.0
  0.0    0.0    -0.08   0.0    0.0
 -1.049  0.878   0.0    2.322  0.0
  0.0    0.0    -2.322  0.0    0.0
  0.0    0.0     0.0    0.0    0.0

[:, :, 3, 1] =
  0.0    0.0     0.0    -0.136  0.0
  0.0    0.0     0.0    -0.08   0.0
  0.0    0.0     0.0     0.0    2.322
 -1.049  0.878   0.0     0.0    0.0
  0.0    0.0    -2.322   0.0    0.0

[:, :, 4, 1] =
  0.0    0.0    0.0   0.0    -0.136
  0.0    0.0    0.0   0.0    -0.08
  0.0    0.0    0.0   0.0     0.0
  0.0    0.0    0.0   0.0     2.322
 -1.049  0.878  0.0  -2.322   0.0

[:, :, 1, 2] =
  0.0    0.0    0.0   0.0    0.543
  0.0    0.0    0.0   0.0    0.238
  0.0    0.0    0.0   0.0    0.0
  0.0    0.0    0.0   0.0    0.146
 -1.106  0.167  0.0  -0.146  0.0

[:, :, 2, 2] =
 -0.301   0.554   0.0    0.173  0.0
  0.182   0.412   0.0    0.588  0.0
  0.0     0.0     0.0    0.0    1.055
 -0.256  -0.943   0.0   -0.41   0.0
  0.0     0.0    -0.48   0.0    0.0

[:, :, 3, 2] =
  0.0     0.0    0.0   0.0   0.173
  0.0     0.0    0.0   0.0   0.588
  0.0     0.0    0.0   0.0   0.0
  0.0     0.0    0.0   0.0   0.644

 -0.256  -0.943  0.0  -0.89  0.0
 -0.256  -0.943  0.0  -0.89  0.0

[:, :, 4, 2] =
 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  -0.246

[:, :, 1, 3] =
 0.0     0.0    0.0    -0.543   0.0
 0.0     0.0    0.0    -0.238   0.0
 0.0     0.0    0.0     0.0    -0.146
 1.106  -0.167  0.0     0.0     0.0
 0.0     0.0    0.146   0.0     0.0

[:, :, 2, 3] =
 0.0    0.0    -0.173   0.0    0.0
 0.0    0.0    -0.588   0.0    0.0
 0.256  0.943   0.0    -0.644  0.0
 0.0    0.0     0.89    0.0    0.0
 0.0    0.0     0.0     0.0    0.0

[:, :, 3, 3] =
 -0.301  0.554  0.0    0.0    0.0
  0.182  0.412  0.0    0.0    0.0
  0.0    0.0    0.0    0.0    0.41
  0.0    0.0    0.0   -0.164  0.0
  0.0    0.0    0.41   0.0    0.0

[:, :, 4, 3] =
  0.0     0.0    0.0   0.0    0.173
  0.0     0.0    0.0   0.0    0.588
  0.0     0.0    0.0   0.0    0.0
  0.0     0.0    0.0   0.0    0.89
 -0.256  -0.943  0.0  -0.644  0.0

[:, :, 1, 4] =
  0.0    0.0     0.543  0.0    0.0
  0.0    0.0     0.238  0.0    0.0
 -1.106  0.167   0.0    0.146  0.0
  0.0    0.0    -0.146  0.0    0.0
  0.0    0.0     0.0    0.0    0.0

[:, :, 2, 4] =
 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  -0.246  0.0  0.0
 0.0  0.0   0.0    0.0  0.0
 0.0  0.0   0.0    0.0  0.0

[:, :, 3, 4] =
 0.0    0.0    -0.173   0.0   0.0
 0.0    0.0    -0.588   0.0   0.0
 0.256  0.943   0.0    -0.89  0.0
 0.0    0.0     0.644   0.0   0.0
 0.0    0.0     0.0     0.0   0.0

[:, :, 4, 4] =
 -0.301  0.554  0.0    -0.173   0.0
  0.182  0.412  0.0    -0.588   0.0
  0.0    0.0    0.0     0.0    -0.48
  0.256  0.943  0.0    -0.41    0.0
  0.0    0.0    1.055   0.0     0.0
julia> d1 = dim(codomain(t))
25

julia> d2 = dim(domain(t))
16

julia> (matrix = reshape(array, d1, d2)) |> disp
25×16 Array{Float64,2}:
 -0.378   0.0      0.0      0.0      0.0     -0.301   0.0      0.0      0.0      0.0     -0.30
  0.704   0.0      0.0      0.0      0.0      0.182   0.0      0.0      0.0      0.0      0.18
  0.0    -1.049    0.0      0.0      0.0      0.0     0.0      0.0      0.0      0.256    0.0
  0.0     0.0     -1.049    0.0      0.0     -0.256   0.0      0.0      1.106    0.0      0.0
  0.0     0.0      0.0     -1.049   -1.106    0.0    -0.256    0.0      0.0      0.0      0.0
 -1.142   0.0      0.0      0.0      0.0      0.554   0.0      0.0      0.0      0.0      0.55
 -0.118   0.0      0.0      0.0      0.0      0.412   0.0      0.0      0.0      0.0      0.41
  0.0     0.878    0.0      0.0      0.0      0.0     0.0      0.0      0.0      0.943    0.0
  0.0     0.0      0.878    0.0      0.0     -0.943   0.0      0.0     -0.167    0.0      0.0
  0.0     0.0      0.0      0.878    0.167    0.0    -0.943    0.0      0.0      0.0      0.0
  0.0    -0.136    0.0      0.0      0.0      0.0     0.0      0.0      0.0     -0.173    0.0
  0.0    -0.08     0.0      0.0      0.0      0.0     0.0      0.0      0.0     -0.588    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    -2.322    0.0      0.0      0.0      0.0     0.0      0.0      0.0      0.89     0.0
  0.391   0.0     -2.322    0.0      0.0     -0.48    0.0      0.0      0.146    0.0      0.41
  0.0     0.0     -0.136    0.0      0.0      0.173   0.0      0.0     -0.543    0.0      0.0
  0.0     0.0     -0.08     0.0      0.0      0.588   0.0      0.0     -0.238    0.0      0.0
  0.0     2.322    0.0      0.0      0.0      0.0     0.0      0.0      0.0     -0.644    0.0
 -0.391   0.0      0.0      0.0      0.0     -0.41    0.0      0.0      0.0      0.0     -0.16
  0.0     0.0      0.0     -2.322   -0.146    0.0    -0.89     0.0      0.0      0.0      0.0

  0.0     0.0      0.0     -0.136    0.543    0.0     0.173    0.0      0.0      0.0      0.0
  0.0     0.0     0.0    -0.136   0.343   0.0     0.175   0.0     0.0     0.0     0.0
  0.0     0.0     0.0    -0.08    0.238   0.0     0.588   0.0     0.0     0.0     0.0
  0.391   0.0     2.322   0.0     0.0     1.055   0.0     0.0    -0.146   0.0     0.41
  0.0     0.0     0.0     2.322   0.146   0.0     0.644   0.0     0.0     0.0     0.0
  0.0     0.0     0.0     0.0     0.0     0.0     0.0    -0.246   0.0     0.0     0.0
julia> (u = reshape(convert(Array, unitary(codomain(t), fuse(codomain(t)))), d1, d1))
25×25 Array{Float64,2}:
 1.0  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  1.0  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    1.0  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  1.0  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  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.
 0.0  0.0  1.0  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  1.0   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  1.0  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  1.0  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  1.0  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  1.0  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  1.0  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  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.
 0.0  0.0  0.0  0.0   0.577  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  1.0  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  1.0  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  -0.577  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.
 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  1.0  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  1.
 0.0  0.0  0.0  0.0   0.577  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.
 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.
julia> (v = reshape(convert(Array, unitary(domain(t), fuse(domain(t)))), d2, d2)) |> d
16×16 Array{Float64,2}:
 1.0  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    1.0  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  1.0  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  1.0  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    0.999   0.0     0.0    0.0    0.0     0.0
 0.0  0.577  0.0  0.0  0.0  0.0     0.0    0.0     0.0     0.707  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    0.707  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    -0.999  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    -0.707   0.0    0.0    0.0    -0.707
 0.0  0.577  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    0.707  0.0     0.0

 0.0  0.0    0.0  0.0  0.0  0.999   0.0    0.0     0.0     0.0    0.0    0.0     0.0
 0.0  0.0    0.0  0.0  0.0  0.999    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     0.999   0.0
 0.0  0.0    0.0  0.0  0.0  0.0      0.0     0.0     -0.707   0.0     0.0     0.0     0.707
 0.0  0.577  0.0  0.0  0.0  0.0      0.0     0.0      0.0    -0.707   0.0     0.0     0.0
julia> u'*u ≈ I ≈ v'*v
true

julia> (u'*matrix*v) |> disp
25×16 Array{Float64,2}:
 -0.378  -0.522   0.0     0.0     0.0     0.0     0.0     0.0     0.0    -0.0     0.0
  0.704   0.316   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
 -1.142   0.96    0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
 -0.118   0.714   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
  0.678   0.985   0.0    -0.0     0.0     0.0    -0.0     0.0     0.0    -0.0     0.0
  0.0     0.0    -1.049   0.0     0.0    -1.106   0.0     0.0    -0.362   0.0     0.0
  0.0     0.0     0.0    -1.049   0.0     0.0    -1.106   0.0     0.0    -0.362   0.0
  0.0     0.0     0.0     0.0    -1.049   0.0     0.0    -1.106   0.0     0.0    -0.36
  0.0     0.0     0.878   0.0     0.0     0.167   0.0     0.0    -1.334   0.0     0.0
  0.0    -0.0     0.0     0.878   0.0     0.0     0.167   0.0     0.0    -1.334   0.0
  0.0     0.0     0.0     0.0     0.878   0.0     0.0     0.167   0.0     0.0    -1.33
  0.0     0.0    -0.136   0.0     0.0     0.543   0.0     0.0     0.245   0.0     0.0
  0.0    -0.0     0.0    -0.136   0.0     0.0     0.543   0.0     0.0     0.245   0.0
  0.0     0.0     0.0     0.0    -0.136   0.0     0.0     0.543   0.0     0.0     0.24
  0.0     0.0    -0.08    0.0     0.0     0.238   0.0     0.0     0.832   0.0     0.0
  0.0    -0.0     0.0    -0.08    0.0     0.0     0.238   0.0     0.0     0.832   0.0
  0.0     0.0     0.0     0.0    -0.08    0.0     0.0     0.238   0.0     0.0     0.83
  0.0     0.0     3.284   0.0     0.0     0.206   0.0     0.0     1.535   0.0     0.0
  0.0    -0.0     0.0     3.284   0.0     0.0     0.206   0.0     0.0     1.535   0.0
  0.0     0.0     0.0     0.0     3.284   0.0     0.0     0.206   0.0     0.0     1.53
  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     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
julia> # compare with:
       block(t, SU2Irrep(0)) |> disp
5×2 Array{Float64,2}:
 -0.378  -0.522
  0.704   0.316
 -1.142   0.96
 -0.118   0.714
  0.678   0.985
julia> block(t, SU2Irrep(1)) |> disp
5×3 Array{Float64,2}:
 -1.049  -1.106  -0.362
  0.878   0.167  -1.334

 -0.136   0.543   0.245
 -0.08    0.238   0.832
  3.284   0.206   1.535
julia> block(t, SU2Irrep(2)) |> disp
1×1 Array{Float64,2}:
 -0.246
```

Note that the basis transforms `u` and `v` are no longer permutation matrices, but are still unitary. Furthermore, note that they render the tensor block diagonal, but that now every element of the diagonal blocks labeled by `c` comes itself in a tensor product with an identity matrix of size `dim(c)`, i.e. `dim(SU2Irrep(1)) = 3` and `dim(SU2Irrep(2)) = 5`.

Tensor properties

Given a `t::AbstractTensorMap{S,N₁,N₂}`, there are various methods to query its properties. The most important are clearly `codomain(t)` and `domain(t)`. The `space(t)` gives the corresponding `HomSpace`. We can also query `space(t, i)`, the space associated with the `i`th index. For `i ∈ 1:N₁`, this corresponds to `codomain(t, i) = codomain(t)[i]`. For `j = i-N₁ ∈ (1:N₂)`, this corresponds to `dual(domain(t, j)) = dual(domain(t)[j])`.

The total number of indices, i.e. `N₁+N₂`, is given by `numind(t)`, with `N₁ == numout(t)` and `N₂ == numin(t)`, the number of outgoing and incoming indices. There are also the unexported methods `TensorXD.codomainind(t)` and `TensorXD.domainind(t)` which return the tuples `(1, 2, …, N₁)` and `(N₁+1, …, N₁+N₂)`, and are useful for internal purposes. The type parameter `S<:ElementarySpace` can be obtained as `spacetype(t)`; the corresponding sector can directly obtained as `sectortype(t)` and is `Trivial` when `S != GradedSpace`. The underlying field scalars of `S` can also directly be obtained as `field(t)`. This is different from `eltype(t)`, which returns the type of `Number` in the tensor data, i.e. the type parameter `T` in the (subtype of) `DenseMatrix{T}` in which the matrix blocks are stored. Note that during construction, a (one-time) warning is printed if `!(T ⊂ field(S))`. The specific `DenseMatrix{T}` subtype in which the tensor data is stored is obtained as `storagetype(t)`. Each of the methods `numind`, `numout`, `numin`, `TensorXD.codomainind`, `TensorXD.domainind`, `spacetype`, `sectortype`, `field`, `eltype` and `storagetype` work in the type domain as well, i.e. they are encoded in `typeof(t)`.

Finally, there are methods to probe the data, which we already encountered. `blocksectors(t)` returns an iterator over the different coupled sectors that can be obtained from fusing the uncoupled sectors available in the domain, but they must also be obtained from fusing the uncoupled sectors available in the codomain (i.e. it is the intersection of both `blocksectors(codomain(t))` and `blocksectors(domain(t))`). For a specific sector `c ∈ blocksectors(t)`, `block(t, c)` returns the corresponding data. Both are obtained together with `blocks(t)`, which returns an iterator over the pairs `c=>block(t, c)`. Furthermore, there is `fusiontrees(t)` which returns an iterator over splitting-fusion tree pairs `(f₁,f₂)`, for which the corresponding data is given by `t[f₁,f₂]` (i.e. using Base.getindex).

Let's again illustrate these methods with an example, continuing with the tensor `t` from the previous example

```
julia> typeof(t)
TensorMap{GradedSpace{SU2Irrep,TensorXD.SortedVectorDict{SU2Irrep,Int64}},2,2,SU2Irrep

julia> codomain(t)
(Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>2, 1=>1))

julia> domain(t)
(Rep[SU₂](0=>1, 1=>1) ⊗ Rep[SU₂](0=>1, 1=>1)')

julia> space(t,1)
Rep[SU₂](0=>2, 1=>1)

julia> space(t,2)
Rep[SU₂](0=>2, 1=>1)

julia> space(t,3)
Rep[SU₂](0=>1, 1=>1)'

julia> space(t,4)
Rep[SU₂](0=>1, 1=>1)

julia> numind(t)
4

julia> numout(t)
2

julia> numin(t)
2

julia> spacetype(t)
GradedSpace{SU2Irrep,TensorXD.SortedVectorDict{SU2Irrep,Int64}}

julia> sectortype(t)
SU2Irrep

julia> field(t)
ℂ

julia> eltype(t)
Float64


julia> storagetype(t)
julia> storagetype(t)
Array{Float64,2}

julia> blocksectors(t)
3-element Array{SU2Irrep,1}:
 0
 1
 2

julia> blocks(t)
TensorXD.SortedVectorDict{SU2Irrep,Array{Float64,2}} with 3 entries:
  Irrep[SU₂](0) => [-0.378031 -0.522084; 0.704204 0.316777; … ; -0.118978 0.714…
  Irrep[SU₂](1) => [-1.04988 -1.10678 -0.362816; 0.878256 0.167477 -1.33422; … …
  Irrep[SU₂](2) => [-0.246326]

julia> block(t, first(blocksectors(t)))
5×2 Array{Float64,2}:
 -0.378031  -0.522084
  0.704204   0.316777
 -1.14276    0.960896
 -0.118978   0.714316
  0.678509   0.985504

julia> fusiontrees(t)
TensorXD.TensorKeyIterator{SU2Irrep,FusionTree{SU2Irrep,2,0,1,Nothing},FusionTree{SU2I

julia> f1, f2 = first(fusiontrees(t))
(FusionTree{Irrep[SU₂]}((1, 1), 0, (false, false), ()), FusionTree{Irrep[SU₂]}((0, 0),

julia> t[f1,f2]
1×1×1×1 Strided.StridedView{Float64,4,Array{Float64,1},typeof(identity)}:
[:, :, 1, 1] =
 0.6785089624817361
```

Reading and writing tensors: `Dict` conversion

There are no custom or dedicated methods for reading, writing or storing `TensorMaps`, however, there is the possibility to convert a `t::AbstractTensorMap` into a `Dict`, simply as `convert(Dict, t)`. The backward conversion `convert(TensorMap, dict)` will return a tensor that is equal to `t`, i.e. `t == convert(TensorMap, convert(Dict, t))`.

This conversion relies on that the string represenation of objects such as `VectorSpace`, `FusionTree` or `Sector` should be such that it represents valid code to recreate the object. Hence, we store information about the domain and codomain of the tensor, and the sector associated with each data block, as a `String` obtained with `repr`. This provides the flexibility to still change the internal structure of such objects, without this breaking the ability to load older data files. The resulting dictionary can then be stored using any of the provided Julia packages such as JLD.jl, JLD2.jl, BSON.jl, JSON.jl, ...

Vector space and linear algebra operations

`AbstractTensorMap` instances `t` represent linear maps, i.e. homomorphisms in a 𝕜-linear category, just like matrices. To a large extent, they follow the interface of `Matrix` in Julia's `LinearAlgebra` standard library. Many methods from `LinearAlgebra` are (re)exported by TensorXD.jl, and can then us be used without `using LinearAlgebra` explicitly. In all of the following methods, the implementation acts directly on the underlying matrix blocks and never needs to perform any basis transforms.

1. Compose tensor maps:

   The `AbstractTensorMap` instances can be composed, provided the domain of the first object coincides with the codomain of the second. Composing tensor maps uses the regular multiplication symbol as in `t = t1*t2`, which is also used for matrix multiplication. TensorXD.jl also supports and exports the mutating method `mul!(t, t1, t2)`.
2. Invert a tensor map:

   We can invert a tensor map using `inv(t)`, though if the domain and codomain are isomorphic, which can be checked by `fuse(codomain(t)) == fuse(domain(t))`. If the inverse is composed with another tensor `t2`, we can use the syntax `t1\t2` or `t2/t1`. However, this syntax also accepts instances `t1` whose domain and codomain are not isomorphic, and then amounts to `pinv(t1)`, the Moore-Penrose pseudoinverse. This, however, is only really justified as minimizing the least squares problem if `spacetype(t) <: EuclideanSpace`.
3. Addition and multiplied by a scalar:

   `AbstractTensorMap` instances behave themselves as vectors (i.e. they are 𝕜-linear) and so they can be multiplied by scalars and, if they live in the same space, i.e. have the same domain and codomain, they can be added to each other. There is also a `zero(t)`, the additive identity, which produces a zero tensor with the same domain and codomain as `t`.
4. Fill and copy:

   `TensorMap` supports basic Julia methods such as `fill!` and `copy!`, as well as `copy(t)` to create a copy with independent data.
5. In-place methods: `axpy!`, `axpby!`, `lmul!`, `rmul!` and `mul!`:

   Aside from basic `+` and `*` operations, TensorXD.jl reexports a number of efficient in-place methods from `LinearAlgebra`, such as `axpy!` (for `y ← α * x + y`), `axpby!` (for `y ← α * x + β * y`), `lmul!` and `rmul!` (for `y ← α*y` and `y ← y*α`, which is typically the same) and `mul!`, which can also be used for out-of-place scalar multiplication `y ← α*x`.

6. Norm and dot:

   For `t::AbstractTensorMap{S}` where `S<:EuclideanSpace`, henceforth referred to as a `(Abstract)EuclideanTensorMap`, we can compute `norm(t)`, and for two such instances, the inner product `dot(t1, t2)`, provided `t1` and `t2` have the same domain and codomain.

7. Normalize:

   For `(Abstract)EuclideanTensorMap`, `normalize(t)` and `normalize!(t)` return a scaled version of `t` with unit norm. These operations should also exist for `S<:InnerProductSpace`, but requires an interface for defining a custom inner product in these spaces. Currently, there is no concrete subtype of `InnerProductSpace` that is not a subtype of `EuclideanSpace`. In particular, `CartesianSpace`, `ComplexSpace` and `GradedSpace` are all subtypes of `EuclideanSpace`.

8. Adjoint:

   With instances `t::AbstractEuclideanTensorMap` there is associated an adjoint operation, given by `adjoint(t)` or simply `t'`, such that `domain(t') == codomain(t)` and `codomain(t') == domain(t)`. Note that for an instance `t::TensorMap{S,N₁,N₂}`, `t'` is simply stored in a wrapper called `AdjointTensorMap{S,N₂,N₁}`, which is another subtype of `AbstractTensorMap`. This should be mostly unvisible to the user, as all methods should work for this type as well. It can be hard to reason about the index order of `t'`, i.e. index `i` of `t` appears in `t'` at index position `j = TensorXD.adjointtensorindex(t, i)`, where the latter method is typically not necessary and hence unexported. There is also a plural `TensorXD.adjointtensorindices` to convert multiple indices at once. Note that, because the adjoint interchanges domain and codomain, we have `space(t', j) == space(t, i)'`.

9. Equal and approximate:

   `AbstractTensorMap` instances can be tested for exact (`t1 == t2`) or approximate (`t1 ≈ t2`) equality, though the latter requires `norm` can be computed.

10. Multiplicative identity:

    When tensor map instances are endomorphisms, i.e. they have the same domain and codomain, there is a multiplicative identity which can be obtained as `one(t)` or `one!(t)`, where the latter overwrites the contents of `t`. The multiplicative identity on a space `V` can also be obtained using `id(A, V)`, such that for a general homomorphism `t`, we have `t == id(codomain(t))*t == t*id(domain(t))`.

11. Trace and exp:

    For case of endomorphisms `t`, we can compute the trace via `tr(t)` and exponentiate them using `exp(t)`, or if the contents of `t` can be destroyed in the process, `exp!(t)`.

12. Tensor product:

    The tensor product of two `TensorMap` instances `t1` and `t2` is obtained as `t1 ⊗ t2` and results in a new `TensorMap` with `codomain(t1⊗t2) = codomain(t1) ⊗ codomain(t2)` and `domain(t1⊗t2) =`

`domain(t1) ⊗ domain(t2)`.

13. catdomain and catcodomain:

    If we have two `TensorMap{S,N,1}` instances `t1` and `t2` with the same codomain, we can combine them in a way that is analogous to `hcat`, i.e. we stack them such that the new tensor `catdomain(t1, t2)` has also the same codomain, but has a domain which is `domain(t1) ⊕ domain(t2)`. Similarly, if `t1` and `t2` are of type `TensorMap{S,1,N}` and have the same domain, the operation `catcodomain(t1, t2)` results in a new tensor with the same domain and a codomain given by `codomain(t1) ⊕ codomain(t2)`, which is the analogy of `vcat`. Note that direct sum only makes sense between `ElementarySpace` objects, i.e. there is no way to give a tensor product meaning to a direct sum of tensor product spaces.

Time for some more examples:

```
julia> t == t + zero(t) == t*id(domain(t)) == id(codomain(t))*t
true

julia> t2 = TensorMap(randn, ComplexF64, codomain(t), domain(t));

julia> dot(t2, t)
8.298563392934998 + 5.089426778453417im

julia> tr(t2'*t)
8.298563392934998 + 5.089426778453417im

julia> dot(t2, t) ≈ dot(t', t2')
true

julia> dot(t2, t2)
50.98012247811684 + 0.0im

julia> norm(t2)^2
50.98012247811684

julia> t3 = copyto!(similar(t, ComplexF64), t);
ERROR: MethodError: no method matching copyto!(::TensorMap{GradedSpace{SU2Irrep,Tensor
Closest candidates are:
  copyto!(!Matched::AbstractArray, ::Any) at abstractarray.jl:730

julia> t3 == t
false

julia> rmul!(t3, 0.8);


julia> t3 ≈ 0.8*t
```

```
ERROR: SpaceMismatch()

julia> axpby!(0.5, t2, 1.3im, t3);
ERROR: SpaceMismatch()

julia> t3 ≈ 0.5 * t2  +  0.8 * 1.3im * t
ERROR: SpaceMismatch()

julia> t4 = TensorMap(randn, fuse(codomain(t)), codomain(t));

julia> t5 = TensorMap(undef, fuse(codomain(t)), domain(t));

julia> mul!(t5, t4, t) == t4*t
true

julia> inv(t4) * t4 ≈ id(codomain(t))
true

julia> t4 * inv(t4) ≈ id(fuse(codomain(t)))
true

julia> t4 \ (t4 * t) ≈ t
true

julia> t6 = TensorMap(randn, ComplexF64, V1, codomain(t));

julia> numout(t4) == numout(t6) == 1
true

julia> t7 = catcodomain(t4, t6);

julia> foreach(println, (codomain(t4), codomain(t6), codomain(t7)))
ProductSpace(Rep[SU₂](0=>5, 1=>5, 2=>1))
ProductSpace(Rep[SU₂](0=>2, 1=>1))
ProductSpace(Rep[SU₂](0=>7, 1=>6, 2=>1))

julia> norm(t7) ≈ sqrt(norm(t4)^2 + norm(t6)^2)
true

julia> t8 = t4 ⊗ t6;

julia> foreach(println, (codomain(t4), codomain(t6), codomain(t8)))
ProductSpace(Rep[SU₂](0=>5, 1=>5, 2=>1))
ProductSpace(Rep[SU₂](0=>2, 1=>1))
(Rep[SU₂](0=>5, 1=>5, 2=>1) ⊗ Rep[SU₂](0=>2, 1=>1))
```

```
julia> foreach(println, (domain(t4), domain(t6), domain(t8)))
(Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>2, 1=>1))
(Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>2, 1=>1))
(Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>2, 1

julia> norm(t8) ≈ norm(t4)*norm(t6)
true
```

Planar index manipulations

In many cases, the bipartition of tensor indices (i.e. `ElementarySpace` instances) between the codomain and domain is not fixed throughout the different operations that need to be performed on that tensor map, i.e. we want to use the duality to move spaces from domain to codomain and vice versa. Furthermore, we want to use the braiding to reshuffle the order of the indices.

Braiding index manipulations

```
braid(t::AbstractTensorMap{S,N₁,N₂}, levels::NTuple{N₁+N₂,Int},
        p1::NTuple{N₁',Int}, p2::NTuple{N₂',Int})
```

and

```
permute(t::AbstractTensorMap{S,N₁,N₂},
        p1::NTuple{N₁',Int}, p2::NTuple{N₂',Int}; copy = false)
```

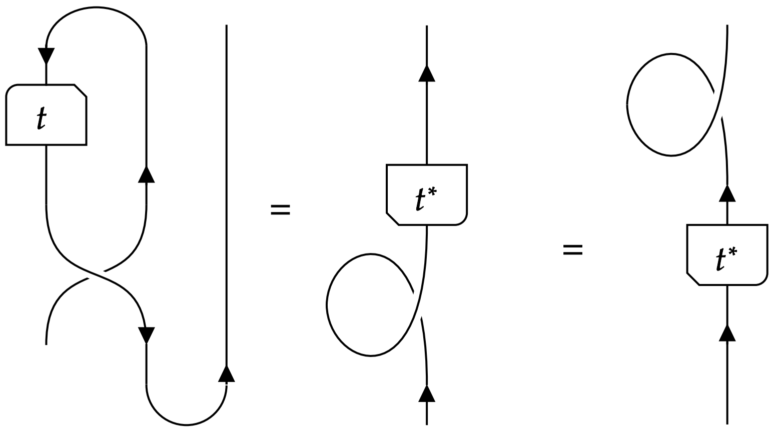
both of which return an instance of `AbstractTensorMap{S,N₁',N₂'}`.

In these methods, `p1` and `p2` specify which of the original tensor indices ranging from `1` to `N₁+N₂` make up the new codomain (with `N₁'` spaces) and new domain (with `N₂'` spaces). Hence, `(p1..., p2...)` should be a valid permutation of `1:(N₁+N₂)`. Note that, throughout TensorXD.jl, permutations are always specified using tuples of `Int`s, for reasons of type stability. For `braid`, we also need to specify `levels` or depths for each of the indices of the original tensor, which determine whether indices will braid over or underneath each other (use the braiding or its inverse). We refer to the section on manipulating fusion trees for more details.

When `BraidingStyle(sectortype(t)) isa SymmetricBraiding`, we can use the simpler interface of `permute`, which does not require the argument `levels`. `permute` accepts a keyword argument `copy`. When `copy == true`, the result will be a tensor with newly allocated data that can independently be modified from that of the input tensor `t`. When `copy` takes the default value `false`, `permute` can try to return the result in a way that it shares its data with the input tensor `t`, though this is only possible in specific cases (e.g. when `sectortype(S) == Trivial` and `(p1..., p2...) = (1:(N₁+N₂)...)`).

Both `braid` and `permute` come in a version where the result is stored in an already existing tensor, i.e. `braid!(tdst, tsrc, levels, p1, p2)` and `permute!(tdst, tsrc, p1, p2)`.

Another operation that belongs und index manipulations is taking the `transpose` of a tensor, i.e. `LinearAlgebra.transpose(t)` and `LinearAlgebra.transpose!(tdst, tsrc)`, both of which are reexported by TensorXD.jl. Note that `transpose(t)` is not simply equal to reshuffling domain and codomain with `braid(t, (1:(N₁+N₂)...), reverse(domainind(tsrc)), reverse(codomainind(tsrc))))`. Indeed, the graphical representation (where we draw the codomain and domain as a single object), makes clear that this introduces an additional (inverse) twist, which is then compensated in the `transpose` implementation.



In categorical language, the reason for this extra twist is that we use the left coevaluation η, but the right evaluation $\tilde{\epsilon}$, when repartitioning the indices between domain and codomain.

There are a number of other index related manipulations. We can apply a twist (or inverse twist) to one of the tensor map indices via `twist(t, i; inv = false)` or `twist!(t, i; inv = false)`. Note that the latter method does not store the result in a new destination tensor, but just modifies the tensor `t` in place. Twisting several indices simultaneously can be obtained by using the defining property

$$\theta_{V\otimes W} = \tau_{W,V} \circ (\theta_W \otimes \theta_V) \circ \tau_{V,W} = (\theta_V \otimes \theta_W) \circ \tau_{W,V} \circ \tau_{V,W}.$$

but is currently not implemented explicitly.

For all sector types `I` with `BraidingStyle(I) == Bosonic()`, all twists are `1` and thus have no effect. Let us start with some examples, in which we illustrate that, albeit `permute` might act highly non-trivial on the fusion trees and on the corresponding data, after conversion to a regular `Array` (when possible), it just acts like `permutedims`

```
julia> domain(t) → codomain(t)
(Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>2, 1=>1)) ← (Rep[SU₂](0=>1, 1=>1) ⊗ Rep[SU₂](0=>1,

julia> ta = convert(Array, t);

julia> t' = permute(t, (1,2,3,4));

julia> domain(t') → codomain(t')
(Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>1, 1=>1)' ⊗ Rep[SU₂](0=>1,

julia> convert(Array, t') ≈ ta
true

julia> t'' = permute(t, (4,2,3),(1,));

julia> domain(t'') → codomain(t'')
(Rep[SU₂](0=>1, 1=>1) ⊗ Rep[SU₂](0=>2, 1=>1) ⊗ Rep[SU₂](0=>1, 1=>1)') ← Rep[SU₂](0=>2,

julia> convert(Array, t'') ≈ permutedims(ta, (4,2,3,1))
true

julia> m
TensorMap(ProductSpace(Rep[SU₂](0=>2, 1=>1)) ← ProductSpace(Rep[SU₂](0=>1, 1=>1))):
* Data for fusiontree FusionTree{Irrep[SU₂]}((0,), 0, (false,), ()) ← FusionTree{Irrep
 -0.6563923539285431
  3.2090440359216394
* Data for fusiontree FusionTree{Irrep[SU₂]}((1,), 1, (false,), ()) ← FusionTree{Irrep
 -0.03366177225273526

julia> transpose(m)
TensorMap(ProductSpace(Rep[SU₂](0=>1, 1=>1)') ← ProductSpace(Rep[SU₂](0=>2, 1=>1)')):
* Data for fusiontree FusionTree{Irrep[SU₂]}((0,), 0, (true,), ()) ← FusionTree{Irrep[
 -0.6563923539285431  3.2090440359216394
* Data for fusiontree FusionTree{Irrep[SU₂]}((1,), 1, (true,), ()) ← FusionTree{Irrep[
 -0.033661772252735256

julia> convert(Array, transpose(t)) ≈ permutedims(ta,(4,3,2,1))
true

julia> dot(t2, t) ≈ dot(transpose(t2), transpose(t))
true

julia> transpose(transpose(t)) ≈ t
true

julia> twist(t, 3) ≈ t
```

```
true

julia> # as twist acts trivially for
       BraidingStyle(sectortype(t))
Bosonic()
```

Note that `transpose` acts like one would expect on a `TensorMap{S,1,1}`. On a `TensorMap{S,N₁,N₂}`, because `transpose` replaces the codomain with the dual of the domain, which has its tensor product operation reversed, this in the end amounts in a complete reversal of all tensor indices when representing it as a plain mutli-dimensional `Array`. Also, note that we have not defined the conjugation of `TensorMap` instances. One definition that one could think of is `conj(t) = adjoint(transpose(t))`. However note that `codomain(adjoint(tranpose(t))) == domain(transpose(t)) == dual(codomain(t))` and similarly `domain(adjoint(tranpose(t))) == dual(domain(t))`, where `dual` of a `ProductSpace` is composed of the dual of the `ElementarySpace` instances, in reverse order of tensor product. This might be very confusing, and as such we leave tensor conjugation undefined. However, note that we have a conjugation syntax within the context of tensor contractions.

To show the effect of `twist`, we now consider a type of sector `I` for which `BraidingStyle{I} != Bosonic()`. In particular, we use `FibonacciAnyon`. We cannot convert the resulting `TensorMap` to an `Array`, so we have to rely on indirect tests to verify our results.

```
julia> V1 = GradedSpace{FibonacciAnyon}(:I=>3,:τ=>2)
Vect[FibonacciAnyon](:I=>3, :τ=>2)

julia> V2 = GradedSpace{FibonacciAnyon}(:I=>2,:τ=>1)
Vect[FibonacciAnyon](:I=>2, :τ=>1)

julia> m = TensorMap(randn, Float32, V1, V2)
TensorMap(ProductSpace(Vect[FibonacciAnyon](:I=>3, :τ=>2)) ← ProductSpace(Vect[Fibonac
* Data for fusiontree FusionTree{FibonacciAnyon}((:I,), :I, (false,), ()) ← FusionTree
 -1.4757175f0 + 0.0f0im    0.7556166f0 + 0.0f0im
 0.43433988f0 + 0.0f0im  -0.14654619f0 + 0.0f0im
  0.8891048f0 + 0.0f0im    -1.370401f0 + 0.0f0im
* Data for fusiontree FusionTree{FibonacciAnyon}((:τ,), :τ, (false,), ()) ← FusionTree
 -0.019544914f0 + 0.0f0im
   0.82385254f0 + 0.0f0im

julia> transpose(m)
TensorMap(ProductSpace(Vect[FibonacciAnyon](:I=>2, :τ=>1)') ← ProductSpace(Vect[Fibona
* Data for fusiontree FusionTree{FibonacciAnyon}((:I,), :I, (true,), ()) ← FusionTree{
 -1.4757175f0 + 0.0f0im   0.43433988f0 + 0.0f0im  0.8891048f0 + 0.0f0im
  0.7556166f0 + 0.0f0im  -0.14654619f0 + 0.0f0im  -1.370401f0 + 0.0f0im
* Data for fusiontree FusionTree{FibonacciAnyon}((:τ,), :τ, (true,), ()) ← FusionTree{

 -0.019544914f0 + 0.0f0im  0.82385254f0 + 0.0f0im
```

```
julia> twist(braid(m, (1,2), (2,), (1,)), 1)
TensorMap(ProductSpace(Vect[FibonacciAnyon](:I=>2, :τ=>1)') ← ProductSpace(Vect[Fibona
* Data for fusiontree FusionTree{FibonacciAnyon}((:I,), :I, (true,), ()) ← FusionTree{
 -1.4757175f0 + 0.0f0im   0.43433988f0 + 0.0f0im  0.8891048f0 + 0.0f0im
  0.7556166f0 + 0.0f0im  -0.14654619f0 + 0.0f0im  -1.370401f0 + 0.0f0im
* Data for fusiontree FusionTree{FibonacciAnyon}((:τ,), :τ, (true,), ()) ← FusionTree{
 -0.019544914f0 - 1.3365192f-10im  0.82385254f0 - 1.250674f-10im

julia> t1 = TensorMap(randn, V1*V2', V2*V1);

julia> t2 = TensorMap(randn, ComplexF64, V1*V2', V2*V1);

julia> dot(t1, t2) ≈ dot(transpose(t1), transpose(t2))
true

julia> transpose(transpose(t1)) ≈ t1
true
```

A final operation that one might expect in this section is to fuse or join indices, and its inverse, to split a given index into two or more indices. For a plain tensor (i.e. with `sectortype(t) == Trivial`) amount to the equivalent of `reshape` on the multidimensional data. However, this represents only one possibility, as there is no canonically unique way to embed the tensor product of two spaces `V₁ ⊗ V₂` in a new space `V = fuse(V₁⊗V₂)`. Such a mapping can always be accompagnied by a basis transform. However, one particular choice is created by the function `isomorphism`, or for `EuclideanSpace` spaces, `unitary`. Hence, we can join or fuse two indices of a tensor by first constructing `u = unitary(fuse(space(t, i) ⊗ space(t, j)), space(t, i) ⊗ space(t, j))` and then contracting this map with indices `i` and `j` of `t`, as explained in the section on contracting tensors. Note, however, that a typical algorithm is not expected to often need to fuse and split indices, as e.g. tensor factorizations can easily be applied without needing to `reshape` or fuse indices first, as explained in the next section.

Tensor factorizations

Eigenvalue decomposition

As tensors are linear maps, they have various kinds of factorizations. Endomorphism, i.e. tensor maps `t` with `codomain(t) == domain(t)`, have an eigenvalue decomposition. For this, we overload both `LinearAlgebra.eigen(t; kwargs...)` and `LinearAlgebra.eigen!(t; kwargs...)`, where the latter destroys `t` in the process. The keyword arguments are the same that are accepted by `LinearAlgebra.eigen(!)` for matrices. The result is returned as `D, V = eigen(t)`, such that `t*V ≈ V*D`. For given `t::TensorMap{S,N,N}`, `V` is a `TensorMap{S,N,1}`, whose codomain corresponds to that of `t`, but whose domain is a single space `S` (or more correctly a `ProductSpace{S,1}`), that corresponds to `fuse(codomain(t))`. The eigenvalues are encoded in `D`, a `TensorMap{S,1,1}`, whose domain and codomain correspond to the domain of `V`. Indeed, we cannot reasonably associate a tensor product structure with the different eigenvalues. Note that `D` stores the eigenvalues on the diagonal of a (collection of) `DenseMatrix` instance(s), as there is currently no dedicated `DiagonalTensorMap` or diagonal storage support.

We also define `LinearAlgebra.ishermitian(t)`, which can only return true for instances of `AbstractEuclideanTensorMap`. In all other cases, as the inner product is not defined, there is no notion of hermiticity (i.e. we are not working in a `†`-category). For instances of `EuclideanTensorMap`, we also define and export the routines `eigh` and `eigh!`, which compute the eigenvalue decomposition under the guarantee (not checked) that the map is hermitian. Hence, eigenvalues will be real and `V` will be unitary with `eltype(V) == eltype(t)`. We also define and export `eig` and `eig!`, which similarly assume that the `TensorMap` is not hermitian (hence this does not require `EuclideanTensorMap`), and always returns complex values eigenvalues and eigenvectors. Like for matrices, `LinearAlgebra.eigen` is type unstable and checks hermiticity at run-time, then falling back to either `eig` or `eigh`.

Orthogonal factorizations

Other factorizations that are provided by TensorXD.jl are orthogonal or unitary in nature, and thus always require a `AbstractEuclideanTensorMap`. However, they don't require equal domain and codomain. Let us first discuss the *singular value decomposition*, for which we define and export the methods `tsvd` and `tsvd!` (where as always, the latter destroys the input).

```
U, Σ, Vʰ, ϵ = tsvd(t; trunc = notrunc(), p::Real = 2,
                        alg::OrthogonalFactorizationAlgorithm = SDD())
```

This computes a (possibly truncated) singular value decomposition of `t::TensorMap{S,N₁,N₂}` (with `S<:EuclideanSpace`), such that `norm(t - U*Σ*Vʰ) ≈ ϵ`, where `U::TensorMap{S,N₁,1}`, `S::TensorMap{S,1,1}`, `Vʰ::TensorMap{S,1,N₂}` and `ϵ::Real`. `U` is an isometry, i.e. `U'*U` approximates the identity, whereas `U*U'` is an idempotent (squares to itself). The same holds for `adjoint(Vʰ)`. The domain of `U` equals the domain and codomain of `Σ` and the codomain of `Vʰ`. In the case of `trunc = notrunc()` (default value, see below), this space is given by `min(fuse(codomain(t)), fuse(domain(t)))`. The singular values are contained in `Σ` and are stored on the diagonal of a (collection of) `DenseMatrix` instance(s), similar to the eigenvalues before.

The keyword argument `trunc` provides a way to control the truncation, and is connected to the keyword argument `p`. The default value `notrunc()` implies no truncation, and thus `ϵ = 0`. Other valid options are

- `truncerr(η::Real)`: truncates such that the `p`-norm of the truncated singular values is smaller than `η` times the `p`-norm of all singular values;

- `truncdim(x::Integer)`: finds the optimal truncation such that the equivalent total dimension of the internal vector space is no larger than `x`;
- `truncspace(W)`: truncates such that the dimension of the internal vector space is smaller than that of `W` in any sector, i.e. with `W₀ = min(fuse(codomain(t)), fuse(domain(t)))` this option will result in `domain(U) == domain(Σ) == codomain(Σ) == codomain(Vᵈ) == min(W, W₀)`;
- `trunbelow(η::Real)`: truncates such that every singular value is larger then `η`; this is different from `truncerr(η)` with `p = Inf` because it works in absolute rather than relative values.

Furthermore, the `alg` keyword can be either `SVD()` or `SDD()` (default), which corresponds to two different algorithms in LAPACK to compute singular value decompositions. The default value `SDD()` uses a divide-and-conquer algorithm and is typically the fastest, but can loose some accuracy. The `SVD()` method uses a QR-iteration scheme and can be more accurate, but is typically slower. Since Julia 1.3, these two algorithms are also available in the `LinearAlgebra` standard library, where they are specified as `LinearAlgebra.DivideAndConquer()` and `LinearAlgebra.QRIteration()`.

Note that we defined the new method `tsvd` (truncated or tensor singular value decomposition), rather than overloading `LinearAlgebra.svd`. We (will) also support `LinearAlgebra.svd(t)` as alternative for `tsvd(t; trunc = notrunc())`, but note that the return values are then given by `U, Σ, V = svd(t)` with `V = adjoint(Vʰ)`.

We also define the following pair of orthogonal factorization algorithms, which are useful when one is not interested in truncating a tensor or knowing the singular values, but only in its image or coimage.

- `Q, R = leftorth(t; alg::OrthogonalFactorizationAlgorithm = QRpos(), kwargs...)`: this produces an isometry `Q::TensorMap{S,N₁,1}` (i.e. `Q'*Q` approximates the identity, `Q*Q'` is an idempotent, i.e. squares to itself) and a general tensor map `R::TensorMap{1,N₂}`, such that `t ≈ Q*R`. Here, the domain of `Q` and thus codomain of `R` is a single vector space of type `S` that is typically given by `min(fuse(codomain(t)), fuse(domain(t)))`.

  The underlying algorithm used to compute this decomposition can be chosen among `QR()`, `QRpos()`, `QL()`, `QLpos()`, `SVD()`, `SDD()`, `Polar()`. `QR()` uses the underlying `qr` decomposition from `LinearAlgebra`, while `QRpos()` (the default) adds a correction to that to make sure that the diagonal elements of `R` are positive. Both result in upper triangular `R`, which are square when `codomain(t) ≾ domain(t)` and wide otherwise. `QL()` and `QLpos()` similarly result in a lower triangular matrices in `R`, but only work in the former case, i.e. `codomain(t) ≾ domain(t)`, which amounts to `blockdim(codomain(t), c) >= blockdim(domain(t), c)` for all `c ∈ blocksectors(t)`.

  One can also use `alg = SVD()` or `alg = SDD()`, with extra keywords to control the absolute (`atol`) or relative (`rtol`) tolerance. We then set `Q=U` and `R=Σ*Vʰ` from the corresponding singular value decomposition, where only these singular values `σ >= max(atol, norm(t)*rtol)` (and corresponding singular vectors in `U`) are kept. More finegrained control on the chosen singular values can be obtained with `tsvd` and its `trunc` keyword.

Finally, `Polar()` sets `Q=U*Vʰ` and `R = (Vʰ)'*Σ*Vʰ`, such that `R` is positive definite; in this case `SDD()` is used to actually compute the singular value decomposition and no `atol` or `rtol` can be provided.

- `L, Q = rightorth(t; alg::OrthogonalFactorizationAlgorithm = QRpos())`: this produces a general tensor map `L::TensorMap{S,N₁,1}` and the adjoint of an isometry `Q::TensorMap{S,1,N₂}`, such that `t ≈ L*Q`. Here, the domain of `L` and thus codomain of `Q` is a single vector space of type `S` that is typically given by `min(fuse(codomain(t)), fuse(domain(t)))`.

  The underlying algorithm used to compute this decomposition can be chosen among `LQ()`, `LQpos()`, `RQ()`, `RQpos()`, `SVD()`, `SDD()`, `Polar()`. `LQ()` uses the underlying `qr` decomposition from `LinearAlgebra` on the transposed data, and leads to lower triangular matrices in `L`; `LQpos()` makes sure the diagonal elements are positive. The matrices `L` are square when `codomain(t) ≿ domain(t)` and tall otherwise. Similarly, `RQ()` and `RQpos()` result in upper triangular matrices in `L`, but only works if `codomain(t) ≿ domain(t)`, i.e. when `blockdim(codomain(t), c) <= blockdim(domain(t), c)` for all `c ∈ blocksectors(t)`.

  One can also use `alg = SVD()` or `alg = SDD()`, with extra keywords to control the absolute (`atol`) or relative (`rtol`) tolerance. We then set `L=U*Σ` and `Q=Vʰ` from the corresponding singular value decomposition, where only these singular values `σ >= max(atol, norm(t)*rtol)` (and corresponding singular vectors in `Vʰ`) are kept. More finegrained control on the chosen singular values can be obtained with `tsvd` and its `trunc` keyword.

  Finally, `Polar()` sets `L = U*Σ*U'` and `Q=U*Vʰ`, such that `L` is positive definite; in this case `SDD()` is used to actually compute the singular value decomposition and no `atol` or `rtol` can be provided.

Furthermore, we can compute an orthonormal basis for the orthogonal complement of the image and of the co-image (i.e. the kernel) with the following methods:

- `N = leftnull(t; alg::OrthogonalFactorizationAlgorithm = QR(), kwargs...)`: returns an isometric `TensorMap{S,N₁,1}` (i.e. `N'*N` approximates the identity) such that `N'*t` is approximately zero.

  Here, `alg` can be `QR()` (`QRpos()` acts identically in this case), which assumes that `t` is full rank in all of its blocks and only returns an orthonormal basis for the missing columns.

  If this is not the case, one can also use `alg = SVD()` or `alg = SDD()`, with extra keywords to control the absolute (`atol`) or relative (`rtol`) tolerance. We then construct `N` from the left singular vectors corresponding to singular values `σ < max(atol, norm(t)*rtol)`.
- `N = rightnull(t; alg::OrthogonalFactorizationAlgorithm = QR(), kwargs...)`: returns a `TensorMap{S,1,N₂}` with isometric adjoint (i.e. `N*N'` approximates the identity) such that `t*N'` is approximately zero.

  Here, `alg` can be `LQ()` (`LQpos()` acts identically in this case), which assumes that `t` is full rank in all of its blocks and only returns an orthonormal basis for the missing rows.

If this is not the case, one can also use `alg = SVD()` or `alg = SDD()`, with extra keywords to control the absolute (`atol`) or relative (`rtol`) tolerance. We then construct `N` from the right singular vectors corresponding to singular values `σ < max(atol, norm(t)*rtol)`.

Note that the methods `leftorth`, `rightorth`, `leftnull` and `rightnull` also come in a form with exclamation mark, i.e. `leftorth!`, `rightorth!`, `leftnull!` and `rightnull!`, which destroy the input tensor `t`.

Factorizations for custom index bipartions

Finally, note that each of the factorizations take a single argument, the tensor map `t`, and a number of keyword arguments. They perform the factorization according to the given codomain and domain of the tensor map. In many cases, we want to perform the factorization according to a different bipartition of the indices. When `BraidingStyle(sectortype(t)) isa SymmetricBraiding`, we can immediately specify an alternative bipartition of the indices of `t` in all of these methods, in the form

```
factorize(t::AbstracTensorMap, pleft::NTuple{N₁',Int}, pright::NTuple{N₂',Int}; kwargs
```

where `pleft` will be the indices in the codomain of the new tensor map, and `pright` the indices of the domain. Here, `factorize` is any of the methods `LinearAlgebra.eigen`, `eig`, `eigh`, `tsvd`, `LinearAlgebra.svd`, `leftorth`, `rightorth`, `leftnull` and `rightnull`. This signature does not allow for the exclamation mark, because it amounts to

```
factorize!(permute(t, pleft, pright; copy = true); kwargs...)
```

where `permute` was introduced and discussed in the previous section. When the braiding is not symmetric, the user should manually apply `braid` to bring the tensor map in proper form before performing the factorization.

Some examples to conclude this section

```
julia> V1 = SU₂Space(0=>2,1/2=>1)
Rep[SU₂](0=>2, 1/2=>1)

julia> V2 = SU₂Space(0=>1,1/2=>1,1=>1)
Rep[SU₂](0=>1, 1/2=>1, 1=>1)

julia> t = TensorMap(randn, V1 ⊗ V1, V2);

julia> U, S, W = tsvd(t);


julia> t ≈ U * S * W
```

```
true

julia> D, V = eigh(t'*t);

julia> D ≈ S*S
true

julia> U'*U ≈ id(domain(U))
true

julia> S
TensorMap(ProductSpace(Rep[SU₂](0=>1, 1/2=>1, 1=>1)) ← ProductSpace(Rep[SU₂](0=>1, 1/2
* Data for fusiontree FusionTree{Irrep[SU₂]}((0,), 0, (false,), ()) ← FusionTree{Irrep
 2.1995412344139096
* Data for fusiontree FusionTree{Irrep[SU₂]}((1/2,), 1/2, (false,), ()) ← FusionTree{I
 2.303460167953715
* Data for fusiontree FusionTree{Irrep[SU₂]}((1,), 1, (false,), ()) ← FusionTree{Irrep
 0.6604301162646513

julia> Q, R = leftorth(t; alg = Polar());

julia> isposdef(R)
true

julia> Q ≈ U*W
true

julia> R ≈ W'*S*W
true

julia> U2, S2, W2, ε = tsvd(t; trunc = truncspace(V1));

julia> W2*W2' ≈ id(codomain(W2))
true

julia> S2
TensorMap(ProductSpace(Rep[SU₂](0=>1, 1/2=>1)) ← ProductSpace(Rep[SU₂](0=>1, 1/2=>1)))
* Data for fusiontree FusionTree{Irrep[SU₂]}((0,), 0, (false,), ()) ← FusionTree{Irrep
 2.1995412344139096
* Data for fusiontree FusionTree{Irrep[SU₂]}((1/2,), 1/2, (false,), ()) ← FusionTree{I
 2.303460167953715

julia> ε ≈ norm(block(S, Irrep[SU₂](1)))*sqrt(dim(Irrep[SU₂](1)))
true
```

```
julia> L, Q = rightorth(t, (1,), (2,3));

julia> codomain(L), domain(L), domain(Q)
(ProductSpace(Rep[SU₂](0=>2, 1/2=>1)), ProductSpace(Rep[SU₂](0=>2, 1/2=>1)), (Rep[SU₂]

julia> Q*Q'
TensorMap(ProductSpace(Rep[SU₂](0=>2, 1/2=>1)) ← ProductSpace(Rep[SU₂](0=>2, 1/2=>1)))
* Data for fusiontree FusionTree{Irrep[SU₂]}((0,), 0, (false,), ()) ← FusionTree{Irrep
 1.0000000000000002     8.287325987765523e-17
 8.287325987765523e-17  1.0000000000000002
* Data for fusiontree FusionTree{Irrep[SU₂]}((1/2,), 1/2, (false,), ()) ← FusionTree{I
 1.0

julia> P = Q'*Q;

julia> P ≈ P*P
true

julia> t′ = permute(t, (1,), (2,3));

julia> t′ ≈ t′ * P
true
```

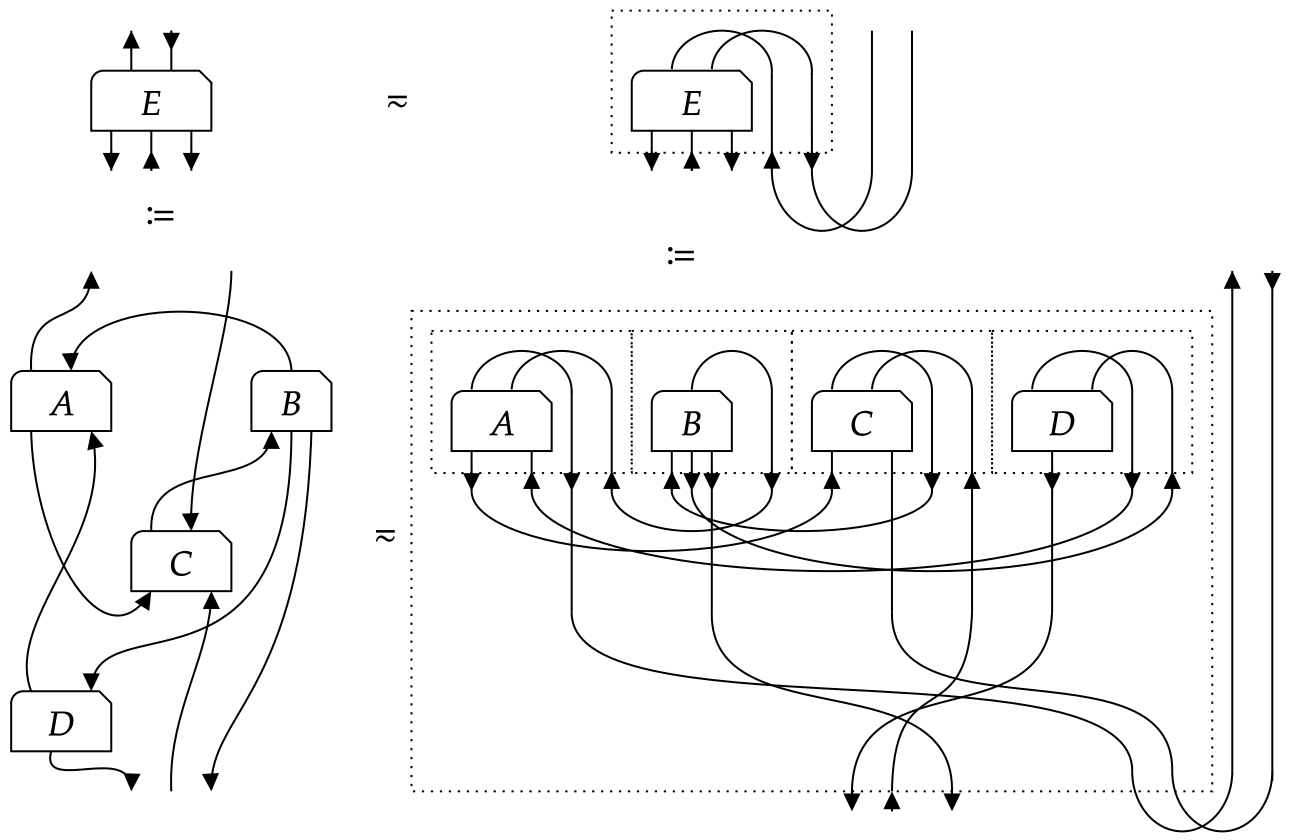
Bosonic tensor contractions and tensor networks

One of the most important operation with tensor maps is to compose them, more generally known as contracting them. As mentioned in the section on category theory, a typical composition of maps in a ribbon category can graphically be represented as a **planar** arrangement of the morphisms (i.e. tensor maps, boxes with lines emanating from top and bottom, corresponding to source and target, i.e. domain and codomain), where the lines connecting the source and targets of the different morphisms should be thought of as ribbons, that can braid over or underneath each other, and that can twist.

Technically, we can embed this diagram in $\mathbb{R} \times [0, 1]$ and attach all the unconnected line endings corresponding objects in the source at some position $(x, 0)$ for $x \in \mathbb{R}$, and all line endings corresponding to objects in the target at some position $(x, 1)$. The resulting morphism is then invariant under **framed three-dimensional isotopy**, i.e. three-dimensional rearrangements of the morphism that respect the rules of boxes connected by ribbons whose open endings are kept fixed. Such a two-dimensional diagram cannot easily be encoded in a single line of code.

However, things simplify when the braiding is symmetric (such that over- and under- crossings become equivalent, i.e. just crossings), and when twists, i.e. self-crossings in this case, are trivial. This amounts to `BraidingStyle(I) == Bosonic()` in the language of TensorXD.jl, and is true for any subcategory of $\mathbf{Vect}$, i.e. ordinary tensors, possibly with some symmetry constraint. The case of $\mathbf{SVect}$ and its subcategories, and more general categories, are discussed below.

In the case of trivial twists, we can deform the diagram such that we first combine every morphism with a number of coevaluations η so as to represent it as a tensor, i.e. with a trivial domain. We can then rearrange the morphism to be all aligned up horizontally, where the original morphism compositions are now being performed by evaluations ϵ. This process will generate a number of crossings and twists. The twists can be omitted because they act trivially. Similarly, double crossings can also be omitted. As a consequence, the diagram, or the morphism it represents, is completely specified by the tensors it is composed of, and which indices between the different tensors are connect, via the evaluation ϵ, and which indices make up the source and target of the resulting morphism. If we also compose the resulting morphisms with coevaluations so that it has a trivial domain, we just have one type of unconnected lines, henceforth called open indices. We sketch such a rearrangement in the following picture



Hence, we can now specify such a tensor diagram, henceforth called a tensor contraction or also tensor network, using a one-dimensional syntax that mimicks abstract index notation and specifies which indices are connected by the evaluation map using Einstein's summation conventation. Indeed, for `BraidingStyle(I) == Bosonic()`, such a tensor contraction can take the same format as if all tensors were just multi-dimensional arrays. For this, we rely on the interface provided by the package TensorOperations.jl.

The above picture would be encoded as

```
@tensor E[a,b,c,d,e] := A[v,w,d,x]*B[y,z,c,x]*C[v,e,y,b]*D[a,w,z]
```

or

```
@tensor E[:] := A[1,2,-4,3]*B[4,5,-3,3]*C[1,-5,4,-2]*D[-1,2,5]
```

where the latter syntax is known as NCON-style, and labels the unconnected or outgoing indices with negative integers, and the contracted indices with positive integers.

A number of remarks are in order. TensorOperations.jl accepts both integers and any valid variable name as dummy label for indices, and everything in `[ ]` is not resolved in the current context but interpreted as a dummy label. Here, we label the indices of a `TensorMap`, like `A::TensorMap{S,N₁,N₂}`, in a linear fashion, where the first position corresponds to the first space in `codomain(A)`, and so forth, up to position `N₁`. Index `N₁+1` then corresponds to the first space in `domain(A)`. However, because we have applied the coevaluation η, it actually corresponds to the corresponding dual space, in accordance with the interface of `space(A, i)` that we introduced above, and as indiated by the dotted box around A in the above picture. The same holds for the other tensor maps. Note that our convention also requires that we braid indices that we brought from the domain to the codomain, and so this is only unambiguous for a symmetric braiding, where there is a unique way to permute the indices.

With the current syntax, we create a new object `E` because we use the definition operator `:=`. Furthermore, with the current syntax, it will be a `Tensor`, i.e. it will have a trivial domain, and correspond to the dotted box in the picture above, rather than the actual morphism `E`. We can also directly define `E` with the correct codomain and domain by rather using

```
@tensor E[a b c;d e] := A[v,w,d,x]*B[y,z,c,x]*C[v,e,y,b]*D[a,w,z]
```

or

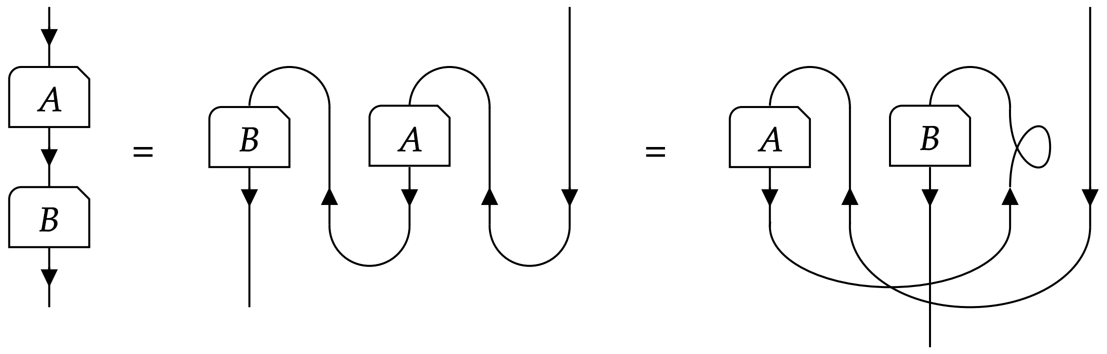
```
@tensor E[(a,b,c);(d,e)] := A[v,w,d,x]*B[y,z,c,x]*C[v,e,y,b]*D[a,w,z]
```

where the latter syntax can also be used when the codomain is empty. When using the assignment operator `=`, the `TensorMap` `E` is assumed to exist and the contents will be written to the currently allocated memory. Note that for existing tensors, both on the left hand side and right hand side, trying to specify the indices in the domain and the codomain seperately using the above syntax, has no effect, as the bipartition of indices are already fixed by the existing object. Hence, if `E` has been created by the previous line of code, all of the following lines are now equivalent

```
@tensor E[(a,b,c);(d,e)] = A[v,w,d,x]*B[y,z,c,x]*C[v,e,y,b]*D[a,w,z]
@tensor E[a,b,c,d,e] = A[v w d;x]*B[(y,z,c);(x,)]*C[v e y; b]*D[a,w,z]
@tensor E[a b; c d e] = A[v; w d x]*B[y,z,c,x]*C[v,e,y,b]*D[a w;z]
```

and none of those will or can change the partition of the indices of `E` into its codomain and its domain.

Two final remarks are in order. Firstly, the order of the tensors appearing on the right hand side is irrelevant, as we can reorder them by using the allowed moves of the Penrose graphical calculus, which yields some crossings and a twist. As the latter is trivial, it can be omitted, and we just use the same rules to evaluate the newly ordered tensor network. For the particular case of matrix matrix multiplication, which also captures more general settings by appropriotely combining spaces into a single line, we indeed find



or thus, the following to lines of code yield the same result

```
@tensor C[i,j] := B[i,k]*A[k,j]
@tensor C[i,j] := A[k,j]*B[i,k]
```

Reordering of tensors can be used internally by the `@tensor` macro to evaluate the contraction in a more efficient manner. In particular, the NCON-style of specifying the contraction gives the user control over the order, and there are other macros, such as `@tensoropt`, that try to automate this process. There is also an `@ncon` macro and `ncon` function, an we recommend reading the manual of TensorOperations.jl to learn more about the possibilities and how they work.

A final remark involves the use of adjoints of tensors. The current framework is such that the user should not be to worried about the actual bipartition into codomain and domain of a given `TensorMap` instance. Indeed, for factorizations one just specifies the requested bipartition via the `factorize(t, pleft, pright)` interface, and for tensor contractions the `@contract` macro figures out the correct manipulations automatically. However, when wanting to use the `adjoint` of an instance `t::TensorMap{S,N₁,N₂}`, the resulting `adjoint(t)` is a `AbstractTensorMap{S,N₂,N₁}` and one need to know the values of `N₁` and `N₂` to know exactly where the `i`th index of `t` will end up in `adjoint(t)`, and hence to know and understand the index order of `t'`. Within the `@tensor` macro, one can instead use `conj()` on the whole index expression so as to be able to use the original index ordering of `t`. Indeed, for matrices of thus, `TensorMap{S,1,1}` instances, this yields exactly the equivalence one expects, namely equivalence between the following to expressions.

```
@tensor C[i,j] := B'[i,k]*A[k,j]
@tensor C[i,j] := conj(B[k,i])*A[k,j]
```

For e.g. an instance `A::TensorMap{S,3,2}`, the following two syntaxes have the same effect within an `@tensor` expression: `conj(A[a,b,c,d,e])` and `A'[d,e,a,b,c]`.

Some examples:

Fermionic tensor contractions

TODO

Anyonic tensor contractions

TODO

« Sectors, representation spaces and fusion trees

Index »