TensorKit.jl

Jutho Haegeman

Apr 3, 2026

Contents

I Home **1**

1 TensorKit.jl 2

1.1 Package summary 2

1.2 Contents of the manual 2

1.3 Library outline 3

1.4 Appendix 4

II Manual **5**

2 Introduction 6

2.1 What is a tensor? 6

2.2 Symmetries and block sparsity 7

3 Tutorial 9

3.1 Cartesian tensors 9

3.2 Complex tensors 19

3.3 Symmetries 23

4 Vector spaces 34

4.1 Fields 34

4.2 Elementary spaces 35

4.3 Operations with elementary spaces 39

4.4 Composite spaces 41

4.5 More operations with vector spaces 43

4.6 Space of morphisms 43

5 Symmetries 47

5.1 Symmetries and symmetric tensors 47

5.2 Representation theory and unitary fusion categories 48

6 Sectors 51

6.1 Minimal sector interface 52

6.2 Additional methods 54

6.3 Additional tools 55

6.4 Group representations 56

6.5 Combining different sectors 63

6.6 Defining a new type of sector 66

6.7 Fermionic sectors 67

6.8 Anyons 70

6.9 Further generalizations 71

7 Graded spaces 72

7.1 Implementation details 72
7.2 Constructing instances 73
7.3 Methods 74
7.4 Examples 75
8 Fusion trees 81
8.1 Canonical representation 81
8.2 Manipulations on a fusion tree 85
8.3 Manipulations on a splitting - fusion tree pair 89
8.4 Inspecting fusion trees as tensors 92
9 Constructing tensors and the `TensorMap` type 96
9.1 Storage of tensor data 96
9.2 Constructing tensor maps and accessing tensor data 101
9.3 Tensor properties 125
9.4 Reading and writing tensors: `Dict` conversion 128
10 Manipulating tensors 130
10.1 Vector space and linear algebra operations 130
10.2 Index manipulations 133
10.3 Tensor factorizations 138
10.4 Bosonic tensor contractions and tensor networks 140
10.5 Fermionic tensor contractions 144
10.6 Anyonic tensor contractions 144
III Library **145**
11 Symmetry sectors 146
11.1 Type hierarchy 146
11.2 Useful constants 155
11.3 Methods for characterizing and manipulating `Sector` objects 155
12 Fusion trees 161
13 Type hierarchy 162
13.1 Methods for defining and generating fusion trees 162
13.2 Methods for manipulating fusion trees 162
14 Vector spaces 169
14.1 Type hierarchy 169
14.2 Useful constants 171
14.3 Methods 172
15 Tensors 181
15.1 Type hierarchy 181
15.2 `TensorMap` constructors 182
15.3 `AbstractTensorMap` properties and data access 187

15.4 AbstractTensorMap operations 193
15.5 TensorMap factorizations 199

IV Index **201**
16 Index 202

V Appendix **208**
17 A symmetric tensor deep dive: constructing your first tensor map 209
17.1 Level 0: The transverse-field Ising model 210
17.2 Level 1: The $\mathbb{Z}_2$-symmetric Ising model 211
17.3 Level 2: The U_1 Bose-Hubbard model 221
17.4 Level 3: Fermions and the Kitaev model 227
17.5 Level 4: Non-Abelian symmetries and the quantum Heisenberg model 236
17.6 Level 5: Anyonic Symmetries and the Golden Chain 250
18 Optional introduction to category theory 254
18.1 Categories, functors and natural transformations 254
18.2 Monoidal categories 256
18.3 Duality: rigid, pivotal and spherical categories 258
18.4 Braidings, twists and ribbons 261
18.5 Adjoints and dagger categories 264
18.6 Direct sums, simple objects and fusion categories 265
18.7 Topological data of a unitary pivotal fusion category 267
18.8 Bibliography 274

VI Changelog **275**
19 Changelog 276
19.1 Guidelines for updating this changelog 276
19.2 Unreleased 276
19.3 0.16.3 - 2026-02-22 276
19.4 0.16.2 - 2026-02-10 277
19.5 0.16.1 - 2026-02-05 277
19.6 0.16.0 - 2025-12-08 278
19.7 0.15.3 - 2025-10-30 279
19.8 0.15.2 - 2025-10-28 279
19.9 0.15.1 - 2025-10-09 279
19.10 0.15.0 - 2025-10-03 279
19.11 0.14.0 - 2024-12-19 280
19.12 0.13.0 - 2024-11-24 280

Part I

Home

Chapter 1

TensorKit.jl

A Julia package for large-scale tensor computations, with a hint of category theory.

1.1 Package summary

TensorKit.jl aims to be a generic package for working with tensors as they appear throughout the physical sciences. TensorKit implements a parametric type `Tensor` (which is actually a specific case of the type `TensorMap`) and defines for these types a number of vector space operations (scalar multiplication, addition, norms and inner products), index operations (permutations) and linear algebra operations (multiplication, factorizations). Finally, tensor contractions can be performed using the `@tensor` macro from TensorOperations.jl.

Currently, most effort is oriented towards tensors as they appear in the context of quantum many-body physics and in particular the field of tensor networks. Such tensors often have large dimensions and take on a specific structure when symmetries are present. By employing concepts from category theory, we can represent and manipulate tensors with a large variety of symmetries, including abelian and non-abelian symmetries, fermionic statistics, as well as generalized (a.k.a. non-invertible or anyonic) symmetries.

At the same time, TensorKit.jl focusses on computational efficiency and performance. The underlying storage of a tensor's data can be any `DenseArray` . When the data is stored in main memory (corresponding to `Array`), multiple CPUs can be leveraged as many operations come with multithreaded implementations, either by distributing the different blocks in case of a structured tensor (i.e. with symmetries) or by using multithreading provided by the package Strided.jl. Support for storing and manipulating tensors on Nvidia and AMD GPUs is currently being developed, whereas support for distributed arrays is planned for the future.

1.2 Contents of the manual

- Introduction
  - What is a tensor?
  - Symmetries and block sparsity
- Vector spaces
  - Fields
  - Elementary spaces
  - Operations with elementary spaces
  - Composite spaces
  - More operations with vector spaces
  - Space of morphisms
- Symmetries
  - Symmetries and symmetric tensors

  - Representation theory and unitary fusion categories
- Sectors
  - Minimal sector interface
  - Additional methods
  - Additional tools
  - Group representations
  - Combining different sectors
  - Defining a new type of sector
  - Fermionic sectors
  - Anyons
  - Further generalizations
- Graded spaces
  - Implementation details
  - Constructing instances
  - Methods
  - Examples
- Fusion trees
  - Canonical representation
  - Manipulations on a fusion tree
  - Manipulations on a splitting - fusion tree pair
  - Inspecting fusion trees as tensors
- Constructing tensors and the `TensorMap` type
  - Storage of tensor data
  - Constructing tensor maps and accessing tensor data
  - Tensor properties
  - Reading and writing tensors: `Dict` conversion
- Manipulating tensors
  - Vector space and linear algebra operations
  - Index manipulations
  - Tensor factorizations
  - Bosonic tensor contractions and tensor networks
  - Fermionic tensor contractions
  - Anyonic tensor contractions

1.3 Library outline

- Symmetry sectors
  - Type hierarchy
  - Useful constants
  - Methods for characterizing and manipulating `Sector` objects

- Fusion trees
- Type hierarchy
  - Methods for defining and generating fusion trees
  - Methods for manipulating fusion trees
- Vector spaces
  - Type hierarchy
  - Useful constants
  - Methods
- Tensors
  - Type hierarchy
  - `TensorMap` constructors
  - `AbstractTensorMap` properties and data access
  - `AbstractTensorMap` operations
  - `TensorMap` factorizations

1.4 Appendix

- A symmetric tensor deep dive: constructing your first tensor map
  - Level 0: The transverse-field Ising model
  - Level 1: The $\mathbb{Z}_2$-symmetric Ising model
  - Level 2: The U_1 Bose-Hubbard model
  - Level 3: Fermions and the Kitaev model
  - Level 4: Non-Abelian symmetries and the quantum Heisenberg model
  - Level 5: Anyonic Symmetries and the Golden Chain
- Optional introduction to category theory
  - Categories, functors and natural transformations
  - Monoidal categories
  - Duality: rigid, pivotal and spherical categories
  - Braidings, twists and ribbons
  - Adjoints and dagger categories
  - Direct sums, simple objects and fusion categories
  - Topological data of a unitary pivotal fusion category
  - Bibliography

Part II

Manual

Chapter 2

Introduction

Before providing a typical *user guide* and discussing the implementation of TensorKit.jl on the next pages, let us discuss some of the rationale behind this package.

2.1 What is a tensor?

At the very start we should ponder about the most suitable and sufficiently general definition of a tensor. A good starting point is the following:

- A tensor t is an element from the tensor product of N vector spaces $V_1, V_2, …, V_N$, where N is referred to as the *rank* or *order* of the tensor, i.e.

$$t \in V_1 \otimes V_2 \otimes … \otimes V_N.$$

If you think of a tensor as an object with indices, a rank N tensor has N indices where every index is associated with the corresponding vector space in that it labels a particular basis in that space. We will return to index notation at the very end of this manual.

As the tensor product of vector spaces is itself a vector space, this implies that a tensor behaves as a vector, i.e. tensors from the same tensor product space can be added and multiplied by scalars. The tensor product is only defined for vector spaces over the same field of scalars, e.g. there is no meaning in $\mathbb{R}^5 \otimes \mathbb{C}^3$. When all the vector spaces in the tensor product have an inner product, this also implies an inner product for the tensor product space. It is hence clear that the different vector spaces in the tensor product should have some form of homogeneity in their structure, yet they do not need to be all equal and can e.g. have different dimensions. It goes without saying that defining the vector spaces and their properties will be an important part of the definition of a tensor. As a consequence, this also constitutes a significant part of the implementation, and is discussed in the section on Vector spaces.

Aside from the interpretation of a tensor as a vector, we also want to interpret it as a matrix (or more correctly, a linear map) in order to decompose tensors using linear algebra factorisations (e.g. eigenvalue or singular value decomposition). Henceforth, we use the term "tensor map" as follows:

- A tensor map t is a linear map from a source or *domain* $W_1 \otimes W_2 \otimes … \otimes W_{N_2}$ to a target or *codomain* $V_1 \otimes V_2 \otimes … \otimes V_{N_1}$, i.e.

$$t : W_1 \otimes W_2 \otimes … \otimes W_{N_2} \to V_1 \otimes V_2 \otimes … \otimes V_{N_1}.$$

A *tensor* of rank N is then just a special case of a tensor map with $N_1 = N$ and $N_2 = 0$. A contraction between two tensors is just a composition of linear maps (i.e. matrix multiplication), where the contracted indices correspond to the domain of the first tensor and the codomain of the second tensor.

In order to allow for arbitrary tensor contractions or decompositions, we need to be able to reorganise which vector spaces appear in the domain and the codomain of the tensor map, and in which order. This amounts to defining canonical isomorphisms between the different ways to order and partition the tensor indices (i.e. the vector spaces). For example, a linear map $W \to V$ is often denoted as a rank two tensor in $V \otimes W^*$, where W^* corresponds to the dual space of W. This simple example introduces two new concepts.

1. Typical vector spaces can appear in the domain and codomain in different related forms, e.g. as normal spaces or dual spaces. In fact, the most generic case is that every vector space V has associated with it a dual space V^*, a conjugate space $\overline{V}$ and a conjugate dual space $\overline{V}^*$. The four different vector spaces V, V^*, $\overline{V}$ and $\overline{V}^*$ correspond to the representation spaces of respectively the fundamental, dual or contragredient, complex conjugate and dual complex conjugate representation of the general linear group $GL(V)$. In index notation these spaces are denoted with respectively contravariant (upper), covariant (lower), dotted contravariant and dotted covariant indices. For real vector spaces, the conjugate (dual) space is identical to the normal (dual) space and we only have upper and lower indices, i.e. this is the setting of e.g. general relativity. For (complex) vector spaces with a sesquilinear inner product $\overline{V} \otimes V \to \mathbb{C}$, the inner product allows to define an isomorphism from the conjugate space to the dual space (known as Riesz representation theorem in the more general context of Hilbert spaces). In particular, in spaces with a Euclidean inner product (the setting of e.g. quantum mechanics), the conjugate and dual space are naturally isomorphic (because the dual and conjugate representation of the unitary group are the same). Again we only need upper and lower indices (or kets and bras). Finally, in $\mathbb{R}^d$ with a Euclidean inner product, these four different spaces are all equivalent and we only need one type of index. The space is completely characterized by its dimension d. This is the setting of much of classical mechanics and we refer to such tensors as cartesian tensors and the corresponding space as cartesian space. These are the tensors that can equally well be represented as multidimensional arrays (i.e. using some `AbstractArray{<:Real, N}` in Julia) without loss of structure. The implementation of all of this is discussed in Vector spaces.
2. In the generic case, the identification between maps $W \to V$ and tensors in $V \otimes W^*$ is not an equivalence but an isomorphism, which needs to be defined. Similarly, there is an isomorphism between between $V \otimes W$ and $W \otimes V$ that can be non-trivial (e.g. in the case of fermions / super vector spaces). The correct formalism here is provided by theory of monoidal categories, the details of which are explained in the appendix. Nonetheless, we try to hide these canonical isomorphisms from the user wherever possible, and one does not need to know category theory to be able to use this package.

This brings us to our final (yet formal) definition

- A tensor (map) is a homomorphism between two objects from the category $\mathbf{Vect}$ (or some subcategory thereof). In practice, this will be $\mathbf{FinVect}$, the category of finite dimensional vector spaces. More generally even, our concept of a tensor makes sense, in principle, for any linear (a.k.a. $\mathbf{Vect}$-enriched) monoidal category. For more details, we refer the curious reader to the appendix on Monoidal categories and their properties.

2.2 Symmetries and block sparsity

Physical problems often have some symmetry, i.e. the setup is invariant under the action of a group G which acts on the vector spaces V in the problem according to a certain representation. Having quantum mechanics in mind, TensorKit.jl is so far restricted to unitary representations. A general representation space V can be specified as the number of times every irreducible representation (irrep) a of G appears, i.e.

$$V = \bigoplus_a \mathbb{C}^{n_a} \otimes R_a$$

with R_a the space associated with irrep a of G, which itself has dimension d_a (often called the quantum dimension), and n_a the number of times this irrep appears in V. If the unitary irrep a for $g \in G$ is given by $u_a(g)$, then there exists a specific basis for V such that the group action of G on V is given by the unitary representation

$$u(g) = \bigoplus_a \mathbb{1}_{n_a} \otimes u_a(g)$$

with $\mathbb{1}_{n_a}$ the $n_a \times n_a$ identity matrix. The total dimension of V is given by $\sum_a n_a d_a$.

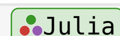
The reason for implementing symmetries is to exploit the computation and memory gains resulting from restricting to tensor maps $t : W_1 \otimes W_2 \otimes \ldots \otimes W_{N_2} \to V_1 \otimes V_2 \otimes \ldots \otimes V_{N_1}$ that are equivariant under the symmetry, i.e. that act as intertwiners between the symmetry action on the domain and the codomain. Indeed, such tensors should be block diagonal because of Schur's lemma, but only after we couple the individual irreps in the spaces W_i to a joint irrep, which is then again split into the individual irreps of the spaces V_i. The basis change from the tensor product of irreps in the (co)domain to the joint irrep is implemented by a sequence of Clebsch–Gordan coefficients, also known as a fusion (or splitting) tree. We implement the necessary machinery to manipulate these fusion trees under index permutations and repartitions for arbitrary groups G. In particular, this fits with the formalism of monoidal categories, and more specifically fusion categories, and only requires the *topological* data of the group, i.e. the fusion rules of the irreps, their quantum dimensions and the F-symbol (6j-symbol or more precisely Racah's W-symbol in the case of SU_2). In particular, we don't actually need the Clebsch–Gordan coefficients themselves (but they can be useful for checking purposes).

Hence, a second major part of TensorKit.jl is the interface and implementation for specifying symmetries, and further details are provided in Sectors, representation spaces and fusion trees.

Chapter 3

Tutorial

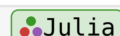
Before discussing at length all aspects of this package, both its usage and implementation, we start with a short tutorial to sketch the main capabilities. Thereto, we start by loading TensorKit.jl



```
julia> using TensorKit
```

3.1 Cartesian tensors

The most important objects in TensorKit.jl are tensors, which we now create with random (normally distributed) entries in the following manner



```
julia> A = randn(ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4)
3×2×4←() TensorMap{Float64, CartesianSpace, 3, 0, Vector{Float64}}:
 codomain: (ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4)
 domain: one(CartesianSpace)
 blocks:
 * Trivial() => 24×1 reshape(view(::Vector{Float64}, 1:24), 24, 1) with eltype
 Float64:
  1.0617991322344207
 -1.406163001897533
 -0.8410062984559351
  0.7197846886334291
  0.10487747409380369
  0.7229831419965808
  1.5669301836273821
  0.08552313918876105
  ⋮
 -0.2523355254587937
 -1.2920529875164615
  0.9329165750538014
 -2.1765363846258374
  0.12402357900807168
 -1.2682461582085764
  0.6139090785686215
```

Note that we entered the tensor size not as plain dimensions, but by specifying the vector space associated with these tensor indices, in this case `ℝ^n` , which can be obtained by typing `\bbR+TAB` . The tensor then lives in the tensor product of the different spaces, which we can obtain by typing `⊗` (i.e. `\otimes+TAB`), although for simplicity also the usual multiplication sign `*` does the job. Note also that `A` is printed as an instance of a parametric type `TensorMap` , which we will discuss below and contains `Tensor` .

Let us briefly sidetrack into the nature of `ℝ^n` :

```julia
julia> V = ℝ^3
ℝ^3

julia> typeof(V)
CartesianSpace

julia> V == CartesianSpace(3)
true

julia> supertype(CartesianSpace)
ElementarySpace

julia> supertype(ElementarySpace)
VectorSpace
```

i.e. `ℝ^n` can also be created without Unicode using the longer syntax `CartesianSpace(n)` . It is a subtype of `ElementarySpace` , with a standard (Euclidean) inner product over the real numbers. Furthermore,

```julia
julia> W = ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4
(ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4)

julia> typeof(W)
ProductSpace{CartesianSpace, 3}

julia> supertype(ProductSpace)
CompositeSpace

julia> supertype(CompositeSpace)
VectorSpace
```

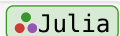
i.e. the tensor product of a number of `CartesianSpace` s is some generic parametric `ProductSpace` type, specifically `ProductSpace{CartesianSpace,N}` for the tensor product of `N` instances of `CartesianSpace` .

Tensors are itself vectors (but not `Vector` s or even `AbstractArray` s), so we can compute linear combinations, provided they live in the same space.

```julia
julia> B = randn(ℝ^3 * ℝ^2 * ℝ^4);

julia> C = 0.5 * A + 2.5 * B
3×2×4←() TensorMap{Float64, CartesianSpace, 3, 0, Vector{Float64}}:
 codomain: (ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4)
 domain: one(CartesianSpace)
 blocks:
 * Trivial() => 24×1 reshape(view(::Vector{Float64}, 1:24), 24, 1) with eltype
 Float64:
 -1.0082670427085532
  5.657916379882947
```

```
 3.2703532683100742
-0.19321757694729502
 0.9111560571389125
 0.7758291874364299
 3.6968572118393905
-1.0115871542380939
 ⋮
 0.3884469109747318
-2.4448995384678067
-0.4281167281200489
-6.371041445705092
-1.7713862021484874
-1.9914386882770105
-0.7168217799759065
```

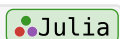
Given that they behave as vectors, they also have a scalar product and norm, which they inherit from the Euclidean inner product on the individual `ℝ^n` spaces:



```julia
julia> scalarBA = dot(B, A)
-1.1179203024321467

julia> scalarAA = dot(A, A)
25.14595282003385

julia> normA² = norm(A)^2
25.145952820033845
```

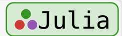
More generally, our tensor objects implement the full interface layed out in VectorInterface.jl.

If two tensors live on different spaces, these operations have no meaning and are thus not allowed



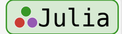
```julia
julia> B′ = randn(ℝ^4 * ℝ^2 * ℝ^3);

julia> space(B′) == space(A)
false

julia> C′ = 0.5 * A + 2.5 * B′
ERROR: SpaceMismatch: (ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4) ← one(CartesianSpace) ≠ (ℝ^4 ⊗ ℝ^2 ⊗ ℝ^3)
← one(CartesianSpace)

julia> scalarBA′ = dot(B′, A)
ERROR: SpaceMismatch: (ℝ^4 ⊗ ℝ^2 ⊗ ℝ^3) ← one(CartesianSpace) ≠ (ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4)
← one(CartesianSpace)
```

However, in this particular case, we can reorder the indices of `B′` to match space of `A`, using the routine `permute` (we deliberately choose not to overload `permutedims` from Julia Base, for reasons that become clear below):



```julia
julia> space(permute(B′, (3, 2, 1))) == space(A)
true
```

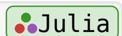
We can contract two tensors using Einstein summation convention, which takes the interface from TensorOperations.jl. TensorKit.jl reexports the `@tensor` macro



```julia
julia> @tensor D[a, b, c, d] := A[a, b, e] * B[d, c, e]
3×2×2×3←() TensorMap{Float64, CartesianSpace, 4, 0, Vector{Float64}}:
 codomain: (ℝ^3 ⊗ ℝ^2 ⊗ ℝ^2 ⊗ ℝ^3)
 domain: one(CartesianSpace)
 blocks:
 * Trivial() => 36×1 reshape(view(::Vector{Float64}, 1:36), 36, 1) with eltype
 Float64:
  2.178397159933017
  1.5549021878201368
  0.05733503200186114
  1.5521177555283738
 -1.3223501041343493
 -2.475833629068416
  2.4683653268153907
 -0.6260978279892304
  ⋮
 -0.6226294456911222
  1.0670899545048482
 -0.3455920341904552
  0.3595757622088753
  0.469344067064465
  0.11445895005496676
 -0.4356791077004852

julia> @tensor d = A[a, b, c] * A[a, b, c]
25.14595282003385

julia> d ≈ scalarAA ≈ normA²
true
```

We hope that the index convention is clear. The `:=` is to create a new tensor `D`, without the `:` the result would be written in an existing tensor `D`, which in this case would yield an error as no tensor `D` exists. If the contraction yields a scalar, regular assignment with `=` can be used.

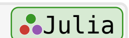
Finally, we can factorize a tensor, creating a bipartition of a subset of its indices and its complement. With a plain Julia `Array`, one would apply `permutedims` and `reshape` to cast the array into a matrix before applying e.g. the singular value decomposition. With TensorKit.jl, one just specifies which indices go to the left (rows) and right (columns) with a tuple of tuples, selecting the respective indices for either side.



```julia
julia> A_matrix = permute(A, ((1, 3), (2,)));
```

```
julia> U, S, Vd = svd_compact(A_matrix);

julia> @tensor A′[a, b, c] := U[a, c, d] * S[d, e] * Vd[e, b];

julia> A ≈ A′
true

julia> U
3×4←2 TensorMap{Float64, CartesianSpace, 2, 1, Vector{Float64}}:
 codomain: (ℝ^3 ⊗ ℝ^4)
 domain: ⊗(ℝ^2)
 blocks:
 * Trivial() => 12×2 reshape(view(::Vector{Float64}, 1:24), 12, 2) with eltype
 Float64:
 -0.245099    0.276044
  0.344216   -0.000785298
  0.218118    0.227157
 -0.357124    0.492429
 -0.0371728  -0.301749
  0.225625   -0.419391
 -0.0079374   0.188042
 -0.316755   -0.34839
  0.19394    -0.108431
  0.316796    0.00881326
 -0.250603   -0.412724
  0.541173    0.154466
```

Note that the `svd_compact` routine returns the decomposition of the linear map as three factors, `U` , `S` and `Vd` , each of them a `TensorMap` , such that `Vd` is already what is commonly called `V'` . Furthermore, observe that `U` is printed differently then `A` , i.e. as a `TensorMap((ℝ^3 ⊗ ℝ^4) ← ProductSpace(ℝ^2))` . What this means is that tensors (or more appropriately, `TensorMap` instances) in TensorKit.jl are always considered to be linear maps between two `ProductSpace` instances, with



```
julia> codomain(U)
(ℝ^3 ⊗ ℝ^4)

julia> domain(U)
⊗(ℝ^2)

julia> codomain(A)
(ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4)

julia> domain(A)
one(CartesianSpace)
```

An instance of `TensorMap` thus represents a linear map from its domain to its codomain, making it an element of the space of homomorphisms between these two spaces. That space is represented using its

own type `HomSpace` in TensorKit.jl, and which admits a direct constructor as well as a unicode alternative using the symbol → (obtained as `\to+TAB` or `\rightarrow+TAB`) or ← (obtained as `\leftarrow+TAB`).

```julia
julia> P = space(U)
(ℝ^3 ⊗ ℝ^4) ← ℝ^2

julia> space(U) == HomSpace(ℝ^3 ⊗ ℝ^4, ℝ^2) == (ℝ^3 ⊗ ℝ^4 ← ℝ^2) == (ℝ^2 → ℝ^3 ⊗
ℝ^4)
true

julia> (codomain(P), domain(P))
((ℝ^3 ⊗ ℝ^4), ⊗(ℝ^2))
```

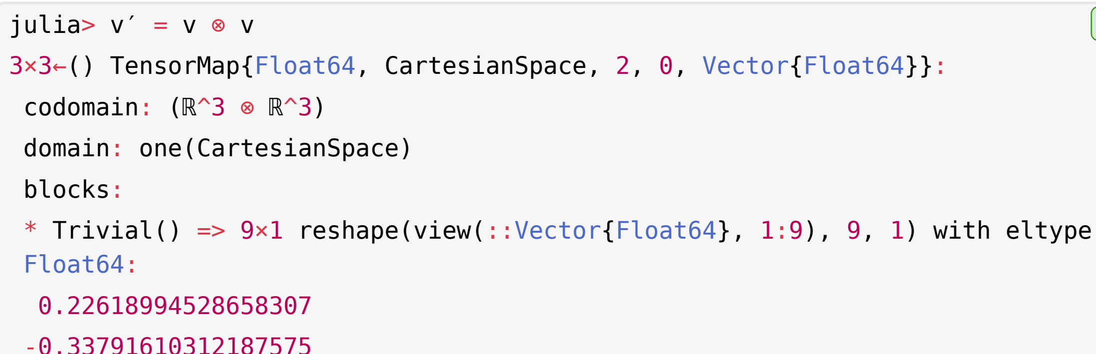
Furthermore, a `Tensor` instance such as `A` is just a specific case of `TensorMap` with an empty domain, i.e. a `ProductSpace{CartesianSpace,0}` instance. Analogously, we can represent a vector `v` and matrix `m` as

```julia
julia> v = randn(ℝ^3)
3←() TensorMap{Float64, CartesianSpace, 1, 0, Vector{Float64}}:
 codomain: ⊗(ℝ^3)
 domain: one(CartesianSpace)
 blocks:
 * Trivial() => 3×1 reshape(view(::Vector{Float64}, 1:3), 3, 1) with eltype
 Float64:
 -0.475594307458135
  0.7105133468226412
 -0.23349351082290992

julia> M₁ = randn(ℝ^4, ℝ^3)
4←3 TensorMap{Float64, CartesianSpace, 1, 1, Vector{Float64}}:
 codomain: ⊗(ℝ^4)
 domain: ⊗(ℝ^3)
 blocks:
 * Trivial() => 4×3 reshape(view(::Vector{Float64}, 1:12), 4, 3) with eltype
 Float64:
 -0.921543  -0.284619   1.01171
  0.207475  -0.458537  -0.0375569
 -1.24272    0.27598   -0.545839
  0.403307   1.95228    1.10185

julia> M₂ = randn(ℝ^4 → ℝ^2) # alternative syntax for randn(ℝ^2, ℝ^4)
2←4 TensorMap{Float64, CartesianSpace, 1, 1, Vector{Float64}}:
 codomain: ⊗(ℝ^2)
 domain: ⊗(ℝ^4)
 blocks:
 * Trivial() => 2×4 reshape(view(::Vector{Float64}, 1:8), 2, 4) with eltype
 Float64:
```

```
 -0.290244  -2.10894  -0.436623  -1.72227
  1.23514    1.43027   0.221529  -1.07639

julia> w = M₁ * v # matrix-vector product
4←() TensorMap{Float64, CartesianSpace, 1, 0, Vector{Float64}}:
 codomain: ⊗(ℝ^4)
 domain: one(CartesianSpace)
 blocks:
 * Trivial() => 4×1 reshape(view(::Vector{Float64}, 1:4), 4, 1) with eltype
 Float64:
 -0.0001740731728065748
 -0.41570136702301047
  0.9145696372259292
  0.9380327694286111

julia> M₃ = M₂ * M₁ # matrix-matrix product
2←3 TensorMap{Float64, CartesianSpace, 1, 1, Vector{Float64}}:
 codomain: ⊗(ℝ^2)
 domain: ⊗(ℝ^3)
 blocks:
 * Trivial() => 2×3 reshape(view(::Vector{Float64}, 1:6), 2, 3) with eltype
 Float64:
 -0.322083  -2.43322  -1.8738
 -1.55091   -3.04764  -0.11104

julia> space(M₃)
ℝ^2 ← ℝ^3
```

Note that for the construction of `M₁` , in accordance with how one specifies the dimensions of a matrix (e.g. `randn(4, 3)`), the first space is the codomain and the second the domain. This is somewhat opposite to the general notation for a function f : domain $\to$ codomain, so that we also support this more mathematical notation, as illustrated in the construction of `M₂` . However, as this is confusing from the perspective of rows and columns, we also support the syntax `codomain ← domain` and actually use this as the default way of printing `HomSpace` instances.

The *matrix-vector* or *matrix-matrix* product can be computed between any two `TensorMap` instances for which the domain of the first matches with the codomain of the second, e.g.



```julia
julia> v′ = v ⊗ v
3×3←() TensorMap{Float64, CartesianSpace, 2, 0, Vector{Float64}}:
 codomain: (ℝ^3 ⊗ ℝ^3)
 domain: one(CartesianSpace)
 blocks:
 * Trivial() => 9×1 reshape(view(::Vector{Float64}, 1:9), 9, 1) with eltype
 Float64:
  0.22618994528658307
 -0.33791610312187575
```

```
  0.1110481845757904
 -0.33791610312187575
  0.5048292160131108
 -0.16590025583615434
  0.1110481845757904
 -0.16590025583615434
  0.054519219596408354

julia> M₁′ = M₁ ⊗ M₁
4×4←3×3 TensorMap{Float64, CartesianSpace, 2, 2, Vector{Float64}}:
 codomain: (ℝ^4 ⊗ ℝ^4)
 domain: (ℝ^3 ⊗ ℝ^3)
 blocks:
 * Trivial() => 16×9 reshape(view(::Vector{Float64}, 1:144), 16, 9) with eltype
 Float64:
  0.849241    0.262289   -0.932338     0.262289   …  -0.287953    1.02356
 -0.191197    0.422562    0.0346103   -0.0590513     -0.463909   -0.0379968
  1.14522    -0.254328    0.503014     0.353703       0.279213   -0.552233
 -0.371665   -1.79911    -1.0154      -0.114789       1.97515     1.11476
 -0.191197   -0.0590513   0.209905     0.422562       0.0106894  -0.0379968
  0.0430457  -0.0951348  -0.00779209  -0.0951348  …   0.0172212   0.00141052
 -0.257834    0.0572589  -0.113248     0.569835      -0.0103649   0.0205
  0.083676    0.405048    0.228606    -0.184932      -0.0733214  -0.0413821
  1.14522     0.353703   -1.25728     -0.254328       0.155356   -0.552233
 -0.257834    0.569835    0.0466728    0.0572589      0.250287    0.0205
  1.54436    -0.342967    0.678327    -0.342967   …  -0.150641    0.29794
 -0.5012     -2.42614    -1.3693       0.111305      -1.06563    -0.601433
 -0.371665   -0.114789    0.408032    -1.79911       -0.313608    1.11476
  0.083676   -0.184932   -0.015147     0.405048      -0.50524    -0.0413821
 -0.5012      0.111305   -0.220141    -2.42614        0.304089   -0.601433
  0.162657    0.787368    0.444385     0.787368   …   2.15112     1.21408

julia> w′ = M₁′ * v′
4×4←() TensorMap{Float64, CartesianSpace, 2, 0, Vector{Float64}}:
 codomain: (ℝ^4 ⊗ ℝ^4)
 domain: one(CartesianSpace)
 blocks:
 * Trivial() => 16×1 reshape(view(::Vector{Float64}, 1:16), 16, 1) with eltype
 Float64:
  3.0301469498317366e-8
  7.236245589771777e-5
 -0.0001592020385044722
 -0.00016328634037091457
  7.236245589771863e-5
  0.17280762654479967
```

```
 -0.3801878484325575
 -0.38994150456385396
 -0.00015920203850447914
 -0.3801878484325575
  0.8364376213355676
  0.8578962896423584
 -0.00016328634037094232
 -0.38994150456385396
  0.8578962896423585
  0.8799054765219096

julia> w′ ≈ w ⊗ w
true
```

Another example involves checking that `U` from the singular value decomposition is a unitary, or at least a (left) isometric tensor

```
julia> codomain(U)
(ℝ^3 ⊗ ℝ^4)

julia> domain(U)
⊗(ℝ^2)

julia> space(U)
(ℝ^3 ⊗ ℝ^4) ← ℝ^2

julia> U' * U # should be the identity on the corresponding domain = codomain
2←2 TensorMap{Float64, CartesianSpace, 1, 1, Vector{Float64}}:
 codomain: ⊗(ℝ^2)
 domain: ⊗(ℝ^2)
 blocks:
 * Trivial() => 2×2 reshape(view(::Vector{Float64}, 1:4), 2, 2) with eltype
 Float64:
 1.0          1.89871e-16
 1.89871e-16  1.0

julia> U' * U ≈ one(U' * U)
true

julia> P = U * U' # should be a projector
3×4←3×4 TensorMap{Float64, CartesianSpace, 2, 2, Vector{Float64}}:
 codomain: (ℝ^3 ⊗ ℝ^4)
 domain: (ℝ^3 ⊗ ℝ^4)
 blocks:
 * Trivial() => 12×12 reshape(view(::Vector{Float64}, 1:144), 12, 12) with eltype
 Float64:
```

```
  0.136274    -0.0845839    0.00924477  …  -0.0525072   -0.0900015
 -0.0845839    0.118486     0.0749013      -0.0859377    0.186159
  0.00924477   0.0749013    0.0991756      -0.148414     0.153128
  0.223463    -0.123315     0.0339635      -0.113741    -0.117202
 -0.0741851   -0.0125585   -0.0766525       0.133855    -0.0667271
 -0.171071     0.0779932   -0.0460545   …   0.11655      0.0573205
  0.0538533   -0.00287985   0.0409837      -0.0756203    0.0247507
 -0.0185345   -0.108759    -0.148229        0.223169    -0.225234
 -0.0774664    0.0668426    0.0176709      -0.00384987   0.0882062
 -0.0752136    0.10904      0.0711009      -0.0830277    0.172803
 -0.0525072   -0.0859377   -0.148414    …   0.233143    -0.199372
 -0.0900015    0.186159     0.153128       -0.199372     0.316728

julia> P * P ≈ P
true
```

Here, the adjoint of a `TensorMap` results in a new tensor map (actually a simple wrapper of type `AdjointTensorMap <: AbstractTensorMap`) with domain and codomain interchanged.

Our original tensor `A` living in `ℝ^4 * ℝ^2 * ℝ^3` is isomorphic to e.g. a linear map `ℝ^3 → ℝ^4 * ℝ^2`. This is where the full power of `permute` emerges. It allows to specify a permutation where some indices go to the codomain, and others go to the domain, as in

```julia
julia> A2 = permute(A, ((1, 2), (3,)))
3×2←4 TensorMap{Float64, CartesianSpace, 2, 1, Vector{Float64}}:
 codomain: (ℝ^3 ⊗ ℝ^2)
 domain: ⊗(ℝ^4)
 blocks:
 * Trivial() => 6×4 reshape(view(::Vector{Float64}, 1:24), 6, 4) with eltype
 Float64:
  1.0618     1.56693     0.0737458  -1.29205
 -1.40616    0.0855231   1.21726     0.932917
 -0.841006  -1.01376    -0.816      -2.17654
  0.719785   1.30894     0.539818    0.124024
  0.104877  -0.881777   -1.10331    -1.26825
  0.722983  -1.13924    -0.252336    0.613909

julia> codomain(A2)
(ℝ^3 ⊗ ℝ^2)

julia> domain(A2)
⊗(ℝ^4)
```

In fact, this was already what we used in `svd_compact(A_matrix)` to create the matricized tensor `A_matrix`, and where `svd_compact(A::AbstractTensorMap)` will just compute the singular value decomposition according to the given codomain and domain of `A`.

Note, finally, that the `@tensor` macro treats all indices at the same footing and thus does not distinguish between codomain and domain. The linear numbering is first all indices in the codomain, followed by all indices in the domain. However, when `@tensor` creates a new tensor (i.e. when using `:=`), the default syntax always creates a `Tensor` , i.e. with all indices in the codomain.

```julia
julia> @tensor A′[a, b, c] := U[a, c, d] * S[d, e] * Vd[e, b];

julia> codomain(A′)
(ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4)

julia> domain(A′)
one(CartesianSpace)

julia> @tensor A2′[(a, b); (c,)] := U[a, c, d] * S[d, e] * Vd[e, b];

julia> codomain(A2′)
(ℝ^3 ⊗ ℝ^2)

julia> domain(A2′)
⊗(ℝ^4)

julia> @tensor A2′′[a b; c] := U[a, c, d] * S[d, e] * Vd[e, b];

julia> A2 ≈ A2′ == A2′′
true
```

As illustrated for `A2′` and `A2′′` , additional syntax is available that enables one to immediately specify the desired codomain and domain indices.

3.2 Complex tensors

For applications in e.g. quantum physics, we of course want to work with complex tensors. Trying to create a complex-valued tensor with `CartesianSpace` indices is of course somewhat contrived and prints a (one-time) warning

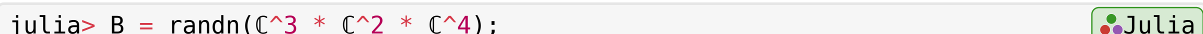
```julia
julia> A = randn(ComplexF64, ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4)
┌ Warning: scalartype(data) = ComplexF64 ⊈ ℝ)
└ @ TensorKit ~/Desktop/undergrad/10-26spring/code/TensorKit.jl/src/tensors/
tensor.jl:29
3×2×4←() TensorMap{ComplexF64, CartesianSpace, 3, 0, Vector{ComplexF64}}:
 codomain: (ℝ^3 ⊗ ℝ^2 ⊗ ℝ^4)
 domain: one(CartesianSpace)
 blocks:
 * Trivial() => 24×1 reshape(view(::Vector{ComplexF64}, 1:24), 24, 1) with eltype
 ComplexF64:
 -0.15629429986835802 - 0.5618629434109697im
  -0.9166250298886501 + 0.49534472842858707im
```

```
  0.08051076456940236 - 0.596820175385897im
 -0.2148961772602218 - 0.5439776733184558im
  0.4036395812763889 + 0.1601311554437251im
  0.15853914830269739 + 0.7878689013866557im
  1.1659234887941723 + 0.2950682852859296im
  0.17336404293596294 + 0.013585164027189444im
                        ⋮
  0.7323362400741038 + 0.820030792418698im
 -1.0533056926288102 - 0.33949521452127407im
 -0.8712145132134608 + 0.66957209995329994im
 -0.6812053699273473 - 0.7858543653992227im
-0.06904364443635748 - 1.2800150531585035im
  0.1826458298011223 - 0.6062632871212519im
  0.16830613891710722 - 0.999574656494328im
```

although most of the above operations will work in the expected way (at your own risk). Indeed, we instead want to work with complex vector spaces

```
julia> A = randn(ComplexF64, ℂ^3 ⊗ ℂ^2 ⊗ ℂ^4)
3×2×4←() TensorMap{ComplexF64, ComplexSpace, 3, 0, Vector{ComplexF64}}:
 codomain: (ℂ^3 ⊗ ℂ^2 ⊗ ℂ^4)
 domain: one(ComplexSpace)
 blocks:
 * Trivial() => 24×1 reshape(view(::Vector{ComplexF64}, 1:24), 24, 1) with eltype
 ComplexF64:
   0.5146092416051572 + 0.22788399017861252im
   0.9600871229208956 + 1.172267583346366im
   0.3902389737607603 + 1.2405650580264491im
  -0.9468585941568883 - 0.6916785797474199im
 -0.05484819396521613 - 0.13492114995816773im
  0.24621545542516932 + 0.21642818027844862im
  -1.038254029752648 - 0.7510474484036083im
  -0.5731477758325589 - 0.3528742857967327im
                      ⋮
    1.592742421120636 + 1.323064165236199im
  -0.6500468674962719 - 0.15480736487810676im
 -0.03170939581479951 - 0.03414502809402737im
   0.2950064292371491 + 0.05322615487471519im
  -1.2840243037743921 - 0.37264643309493906im
 -0.29392953184695336 - 0.5924354895574936im
   0.5848342679332055 - 1.1127506662022735im
```

where `ℂ` is obtained as `\bbC+TAB` and we also have the non-Unicode alternative `ℂ^n == ComplexSpace(n)`. Most functionality works exactly the same

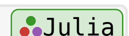


```
julia> B = randn(ℂ^3 * ℂ^2 * ℂ^4);
```

```
julia> C = im * A + (2.5 - 0.8im) * B
3×2×4←() TensorMap{ComplexF64, ComplexSpace, 3, 0, Vector{ComplexF64}}:
 codomain: (ℂ^3 ⊗ ℂ^2 ⊗ ℂ^4)
 domain: one(ComplexSpace)
 blocks:
 * Trivial() => 24×1 reshape(view(::Vector{ComplexF64}, 1:24), 24, 1) with eltype
 ComplexF64:
 -1.3452057548065632 + 0.8721522062861015im
 -0.8969315134865756 + 0.8719795805657626im
  -5.052718335335426 + 1.610128022499633im
  3.1753875832713385 - 1.7416454752845425im
 -1.1275783903610441 + 0.3491516589369317im
   4.201497543260722 - 1.1675207761073652im
   4.948417645446708 - 2.38141249280644im
 -0.4780681375494202 - 0.30724620036179im
                     ⋮
  -2.869644170685917 + 2.0876480228645455im
  -1.544523442283007 - 0.10626100920471548im
   0.280242230627873 - 0.11046050062563012im
  1.5906353678510536 - 0.23102925803509694im
  1.2734504936624722 - 1.5722816031560027im
 -2.1671389831451333 + 0.5891342994178874im
 -0.3735703478754444 + 1.0604569924380751im

julia> scalarBA = dot(B, A)
-1.7575238665498438 - 3.2010808980332026im

julia> scalarAA = dot(A, A)
19.457433588395194 + 0.0im

julia> normA² = norm(A)^2
19.457433588395197

julia> U, S, Vd = svd_compact(permute(A, ((1, 3), (2,))));

julia> @tensor A′[a, b, c] := U[a, c, d] * S[d, e] * Vd[e, b];

julia> A′ ≈ A
true

julia> permute(A, ((1, 3), (2,))) ≈ U * S * Vd
true
```

However, trying the following

```
julia> @tensor D[a, b, c, d] := A[a, b, e] * B[d, c, e]
```



```
ERROR: SpaceMismatch: ⊗((ℂ^4)') ≠ ⊗(ℂ^4)

julia> @tensor d = A[a, b, c] * A[a, b, c]
ERROR: SpaceMismatch: ((ℂ^3)' ⊗ (ℂ^2)' ⊗ (ℂ^4)') ≠ (ℂ^3 ⊗ ℂ^2 ⊗ ℂ^4)
```

we obtain `SpaceMismatch` errors. The reason for this is that, with `ComplexSpace` , an index in a space `ℂ^n` can only be contracted with an index in the dual space `dual(ℂ^n) == (ℂ^n)'` . Because of the complex Euclidean inner product, the dual space is equivalent to the complex conjugate space, but not the space itself.

```julia
julia> dual(ℂ^3) == conj(ℂ^3) == (ℂ^3)'
true

julia> (ℂ^3)' == ℂ^3
false

julia> @tensor d = conj(A[a, b, c]) * A[a, b, c]
19.4574335883952 + 0.0im

julia> d ≈ normA²
true
```

This might seem overly strict or puristic, but we believe that it can help to catch errors, e.g. unintended contractions. In particular, contracting two indices both living in `ℂ^n` would represent an operation that is not invariant under arbitrary unitary basis changes.

It also makes clear the isomorphism between linear maps `ℂ^n → ℂ^m` and tensors in `ℂ^m ⊗ (ℂ^n)'` :

```julia
julia> m = randn(ComplexF64, ℂ^3, ℂ^4)
3←4 TensorMap{ComplexF64, ComplexSpace, 1, 1, Vector{ComplexF64}}:
 codomain: ⊗(ℂ^3)
 domain: ⊗(ℂ^4)
 blocks:
 * Trivial() => 3×4 reshape(view(::Vector{ComplexF64}, 1:12), 3, 4) with eltype
 ComplexF64:
  0.375797-0.943312im   -0.31072+0.111498im  …    0.42166-0.560869im
 -0.663818+0.202314im   0.826952+0.442905im     -0.417863-0.738527im
 -0.238677+0.595073im  -0.328861+1.04242im       0.897144-1.07042im

julia> m2 = permute(m, ((1, 2), ()))
3×4←() TensorMap{ComplexF64, ComplexSpace, 2, 0, Vector{ComplexF64}}:
 codomain: (ℂ^3 ⊗ (ℂ^4)')
 domain: one(ComplexSpace)
 blocks:
 * Trivial() => 12×1 reshape(view(::Vector{ComplexF64}, 1:12), 12, 1) with eltype
 ComplexF64:
 0.37579714159441685 - 0.9433121804956824im
 -0.6638178499566648 + 0.2023140992991712im
```

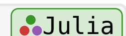
```
 -0.2386768202157103 + 0.59507310637002222im
 -0.3107197765839294 + 0.1114983953418227im
  0.8269516881763159 + 0.4429047701993804im
 -0.3288605633704565 + 1.0424247894859886im
 -0.6091670288395012 - 1.5257251630871036im
 0.16842649703030738 - 1.0159298356854651im
  0.8609360797731554 + 0.6443084233258157im
 0.42165979405380877 - 0.5608692229862484im
 -0.4178633762153724 - 0.738527241910575im
  0.8971437760666692 - 1.0704203948046715im

julia> codomain(m2)
(ℂ^3 ⊗ (ℂ^4)')

julia> space(m, 1)
ℂ^3

julia> space(m, 2)
(ℂ^4)'
```

Hence, spaces become their corresponding dual space if they are 'permuted' from the domain to the codomain or vice versa. Also, spaces in the domain are reported as their dual when probing them with `space(A, i)` . Generalizing matrix-vector and matrix-matrix multiplication to arbitrary tensor contractions require that the two indices to be contracted have spaces which are each others dual. Knowing this, all the other functionality of tensors with `CartesianSpace` indices remains the same for tensors with `ComplexSpace` indices.

3.3 Symmetries

So far, the functionality that we have illustrated seems to be just a convenient (or inconvenient?) wrapper around dense multidimensional arrays, e.g. Julia's Base `Array` . More power becomes visible when involving symmetries. With symmetries, we imply that there is some symmetry action defined on every vector space associated with each of the indices of a `TensorMap` , and the `TensorMap` is then required to be equivariant, i.e. it acts as an intertwiner between the tensor product representation on the domain and that on the codomain. By Schur's lemma, this means that the tensor is block diagonal in some basis corresponding to the irreducible representations that can be coupled to by combining the different representations on the different spaces in the domain or codomain. For Abelian symmetries, this does not require a basis change and it just imposes that the tensor has some block sparsity. Let's clarify all of this with some examples.

We start with a simple $\mathbb{Z}_2$ symmetry:



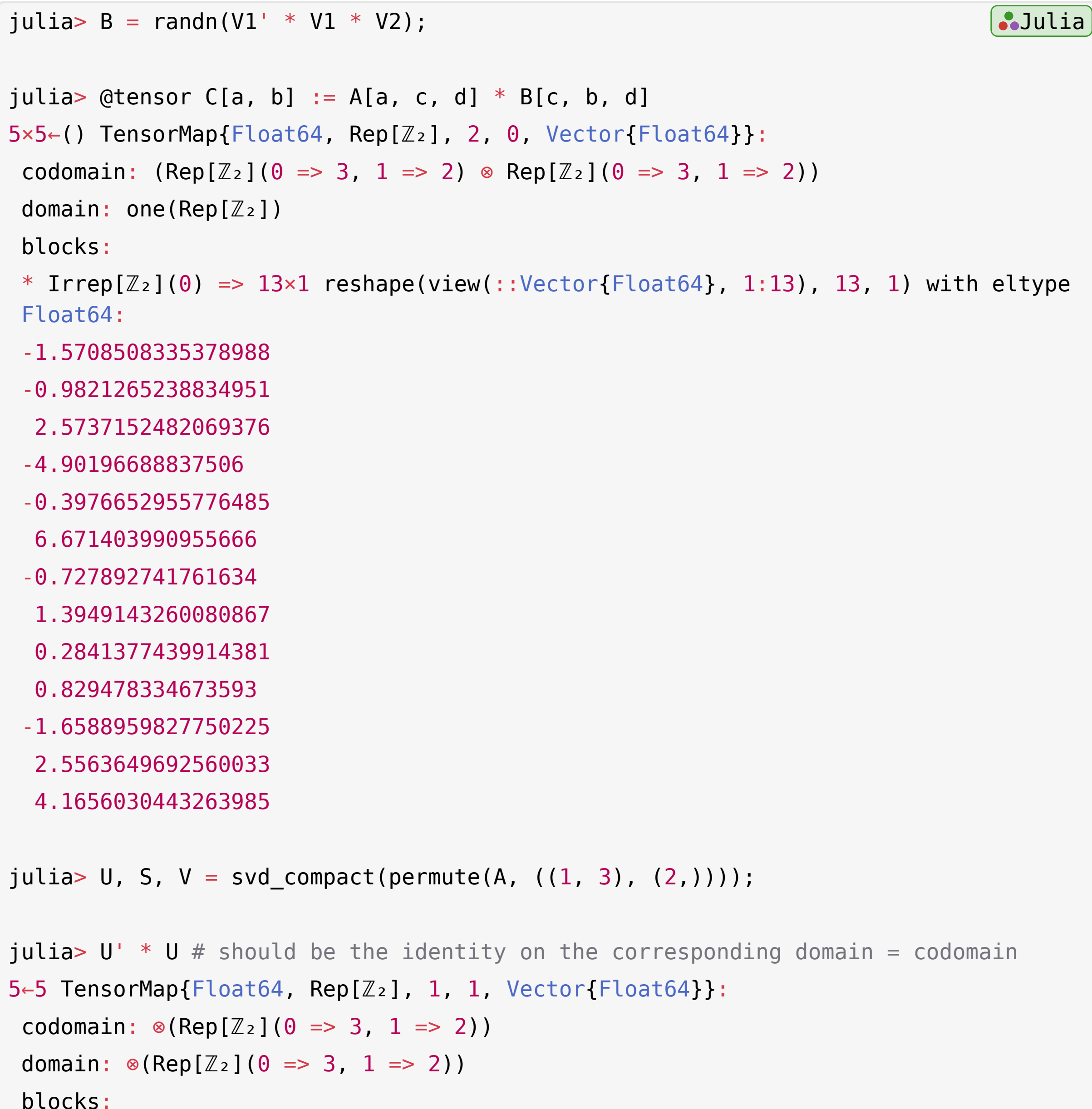

```
julia> V1 = ℤ₂Space(0=>3, 1=>2)
Rep[ℤ₂](…) of dim 5:
 0 => 3
 1 => 2
```

```
julia> dim(V1)
5

julia> V2 = ℤ₂Space(0=>1, 1=>1)
Rep[ℤ₂](…) of dim 2:
 0 => 1
 1 => 1

julia> dim(V2)
2

julia> A = randn(V1 * V1 * V2')
5×5×2←() TensorMap{Float64, Rep[ℤ₂], 3, 0, Vector{Float64}}:
 codomain: (Rep[ℤ₂](0 => 3, 1 => 2) ⊗ Rep[ℤ₂](0 => 3, 1 => 2) ⊗ Rep[ℤ₂](0 => 1, 1
 => 1)')
 domain: one(Rep[ℤ₂])
 blocks:
 * Irrep[ℤ₂](0) => 25×1 reshape(view(::Vector{Float64}, 1:25), 25, 1) with eltype
 Float64:
  0.8005897560575018
  0.19356097062816907
 -0.9796385442428749
 -0.009474900443217521
 -0.7725966110716952
  0.3058421055561289
  0.16366171028073323
 -0.7265984829933455
  ⋮
 -0.2758426730857543
  1.8959077411364538
  0.20447897580863056
 -2.4046891323765394
  0.47742661093525146
 -0.7054678938331744
 -0.8260702984411576

julia> convert(Array, A)
5×5×2 Array{Float64, 3}:
[:, :, 1] =
  0.80059   -0.0094749   0.163662   0.0        0.0
  0.193561  -0.772597   -0.726598   0.0        0.0
 -0.979639   0.305842    0.183926   0.0        0.0
  0.0        0.0         0.0       -0.073064   0.042131
  0.0        0.0         0.0        0.612553  -0.874046
```

```
[:, :, 2] =
 0.0         0.0         0.0         1.89591    0.477427
 0.0         0.0         0.0         0.204479  -0.705468
 0.0         0.0         0.0        -2.40469   -0.82607
 1.5413     -0.216983    0.818108    0.0        0.0
 0.803724   -1.21854    -0.275843    0.0        0.0
```

Here, we create a 5-dimensional space `V1`, which has a three-dimensional subspace associated with charge 0 (the trivial irrep of $\mathbb{Z}_2$) and a two-dimensional subspace with charge 1 (the non-trivial irrep). Similar for `V2`, where both subspaces are one-dimensional. Representing the tensor as a dense `Array`, we see that it is zero in those regions where the charges don't add to zero (modulo 2). Of course, the `Tensor(Map)` type in TensorKit.jl won't store these zero blocks, and only stores the non-zero information, which we can recognize also in the full `Array` representation.

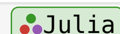
From there on, the resulting tensors support all of the same operations as the ones we encountered in the previous examples.



```julia
julia> B = randn(V1' * V1 * V2);

julia> @tensor C[a, b] := A[a, c, d] * B[c, b, d]
5×5←() TensorMap{Float64, Rep[ℤ₂], 2, 0, Vector{Float64}}:
 codomain: (Rep[ℤ₂](0 => 3, 1 => 2) ⊗ Rep[ℤ₂](0 => 3, 1 => 2))
 domain: one(Rep[ℤ₂])
 blocks:
 * Irrep[ℤ₂](0) => 13×1 reshape(view(::Vector{Float64}, 1:13), 13, 1) with eltype
 Float64:
 -1.5708508335378988
 -0.9821265238834951
  2.5737152482069376
 -4.90196688837506
 -0.3976652955776485
  6.671403990955666
 -0.727892741761634
  1.3949143260080867
  0.2841377439914381
  0.829478334673593
 -1.6588959827750225
  2.5563649692560033
  4.1656030443263985

julia> U, S, V = svd_compact(permute(A, ((1, 3), (2,))));

julia> U' * U # should be the identity on the corresponding domain = codomain
5←5 TensorMap{Float64, Rep[ℤ₂], 1, 1, Vector{Float64}}:
 codomain: ⊗(Rep[ℤ₂](0 => 3, 1 => 2))
 domain: ⊗(Rep[ℤ₂](0 => 3, 1 => 2))
 blocks:
```

```
 * Irrep[ℤ₂](0) => 3×3 reshape(view(::Vector{Float64}, 1:9), 3, 3) with eltype
 Float64:
  1.0           -1.79254e-17  7.38764e-17
 -1.79254e-17   1.0           1.76371e-16
  7.38764e-17   1.76371e-16   1.0

 * Irrep[ℤ₂](1) => 2×2 reshape(view(::Vector{Float64}, 10:13), 2, 2) with eltype
 Float64:
 1.0           3.80376e-17
 3.80376e-17   1.0

julia> U' * U ≈ one(U'*U)
true

julia> P = U * U' # should be a projector
5×2←5×2 TensorMap{Float64, Rep[ℤ₂], 2, 2, Vector{Float64}}:
 codomain: (Rep[ℤ₂](0 => 3, 1 => 2) ⊗ Rep[ℤ₂](0 => 1, 1 => 1)')
 domain: (Rep[ℤ₂](0 => 3, 1 => 2) ⊗ Rep[ℤ₂](0 => 1, 1 => 1)')
 blocks:
 * Irrep[ℤ₂](0) => 5×5 reshape(view(::Vector{Float64}, 1:25), 5, 5) with eltype
 Float64:
  0.284372    0.0271177  -0.375561    0.203742   -0.142171
  0.0271177   0.500532   -0.282639   -0.285553    0.296377
 -0.375561   -0.282639    0.62609    -0.0585528   0.0987017
  0.203742   -0.285553   -0.0585528   0.787129    0.202675
 -0.142171    0.296377    0.0987017   0.202675    0.801876

 * Irrep[ℤ₂](1) => 5×5 reshape(view(::Vector{Float64}, 26:50), 5, 5) with eltype
 Float64:
  0.00259388  -0.0399587  -0.012385   -0.0278048   0.00799744
 -0.0399587    0.649557    0.0882717   0.466684    0.0212238
 -0.012385     0.0882717   0.368302    0.0171057  -0.473728
 -0.0278048    0.466684    0.0171057   0.341316    0.0772008
  0.00799744   0.0212238  -0.473728    0.0772008   0.638231

julia> P * P ≈ P
true
```

We also support other abelian symmetries, e.g.



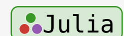
```
julia> V = U₁Space(0 => 2, 1 => 1, -1 => 1)
Rep[U₁](…) of dim 4:
  0 => 2
  1 => 1
 -1 => 1
```

```
julia> dim(V)
4

julia> A = randn(V * V, V)
4×4←4 TensorMap{Float64, Rep[U₁], 2, 1, Vector{Float64}}:
 codomain: (Rep[U₁](0 => 2, 1 => 1, -1 => 1) ⊗ Rep[U₁](0 => 2, 1 => 1, -1 => 1))
 domain: ⊗(Rep[U₁](0 => 2, 1 => 1, -1 => 1))
 blocks:
 * Irrep[U₁](0) => 6×2 reshape(view(::Vector{Float64}, 1:12), 6, 2) with eltype
 Float64:
 -0.828192   -0.298058
 -0.620378   -1.02021
  ⋮
  0.0967386  -0.0336234
 -1.17931    -1.21248

 * Irrep[U₁](1) => 4×1 reshape(view(::Vector{Float64}, 13:16), 4, 1) with eltype
 Float64:
  1.616496888623259
 -0.6058467311453105
  1.2395023334052329
  1.659661469542189

 *   …   [output of 1 more block(s) truncated]

julia> dim(A)
20

julia> convert(Array, A)
4×4×4 Array{Float64, 3}:
[:, :, 1] =
 -0.828192  -2.07614   0.0          0.0
 -0.620378  -0.594258  0.0          0.0
  0.0        0.0       0.0         -1.17931
  0.0        0.0       0.0967386    0.0

[:, :, 2] =
 -0.298058  -0.239944   0.0          0.0
 -1.02021    0.296593   0.0          0.0
  0.0        0.0        0.0         -1.21248
  0.0        0.0       -0.0336234    0.0

[:, :, 3] =
 0.0      0.0       1.2395    0.0
 0.0      0.0       1.65966   0.0
```

```
 1.6165  -0.605847  0.0      0.0
 0.0      0.0       0.0      0.0

[:, :, 4] =
 0.0       0.0       0.0   0.155354
 0.0       0.0       0.0  -0.0685022
 0.0       0.0       0.0   0.0
 0.476869  0.696621  0.0   0.0

julia> V = Rep[U₁ × ℤ₂]((0, 0) => 2, (1, 1) => 1, (-1, 0) => 1)
Rep[U₁ × ℤ₂](…) of dim 4:
  (0, 0) => 2
  (1, 1) => 1
 (-1, 0) => 1

julia> dim(V)
4

julia> A = randn(V * V, V)
4×4←4 TensorMap{Float64, Rep[U₁ × ℤ₂], 2, 1, Vector{Float64}}:
 codomain: (Rep[U₁ × ℤ₂]((0, 0) => 2, (1, 1) => 1, (-1, 0) => 1) ⊗ Rep[U₁ × ℤ₂]
 ((0, 0) => 2, (1, 1) => 1, (-1, 0) => 1))
 domain: ⊗(Rep[U₁ × ℤ₂]((0, 0) => 2, (1, 1) => 1, (-1, 0) => 1))
 blocks:
 * Irrep[U₁ × ℤ₂](0, 0) => 4×2 reshape(view(::Vector{Float64}, 1:8), 4, 2) with
 eltype Float64:
  1.99699   -0.25596
 -0.34195   -2.19238
  0.906975  -1.14238
 -2.15114   -0.109883

 * Irrep[U₁ × ℤ₂](1, 1) => 4×1 reshape(view(::Vector{Float64}, 9:12), 4, 1) with
 eltype Float64:
 -0.320724775265365
 -1.3682358555624663
  1.0458316106379173
  0.8162870267875519

 * Irrep[U₁ × ℤ₂](-1, 0) => 4×1 reshape(view(::Vector{Float64}, 13:16), 4, 1) with
 eltype Float64:
  1.5687234751401535
  1.1654069873809307
  1.3454710706931752
 -0.40392647250278074
```

```
julia> dim(A)
16

julia> convert(Array, A)
4×4×4 Array{Float64, 3}:
[:, :, 1] =
  1.99699   0.906975  0.0  0.0
 -0.34195  -2.15114   0.0  0.0
  0.0       0.0       0.0  0.0
  0.0       0.0       0.0  0.0

[:, :, 2] =
 -0.25596  -1.14238   0.0  0.0
 -2.19238  -0.109883  0.0  0.0
  0.0       0.0       0.0  0.0
  0.0       0.0       0.0  0.0

[:, :, 3] =
  0.0        0.0      1.04583   0.0
  0.0        0.0      0.816287  0.0
 -0.320725  -1.36824  0.0       0.0
  0.0        0.0      0.0       0.0

[:, :, 4] =
 0.0      0.0      0.0   1.34547
 0.0      0.0      0.0  -0.403926
 0.0      0.0      0.0   0.0
 1.56872  1.16541  0.0   0.0
```

Here, the `dim` of a `TensorMap` returns the number of linearly independent components, i.e. the number of non-zero entries in the case of an abelian symmetry. Also note that we can use `×` (obtained as `\times+TAB`) to combine different symmetry groups. The general space associated with symmetries is a `GradedSpace`, which is parametrized to the type of symmetry. For a group `G`, the fully specified type can be obtained as `Rep[G]`, while for more general sectortypes `I` it can be constructed as `Vect[I]`. Furthermore, `ℤ₂Space` (also `Z2Space` as non-Unicode alternative) and `U₁Space` (or `U1Space`) are just convenient synonyms, e.g.



```
julia> Rep[U₁](0 => 3, 1 => 2, -1 => 1) == U1Space(-1 => 1, 1 => 2, 0 =>
3)
true

julia> V = U₁Space(1 => 2, 0 => 3, -1 => 1)
Rep[U₁](…) of dim 6:
  0 => 3
  1 => 2
 -1 => 1
```

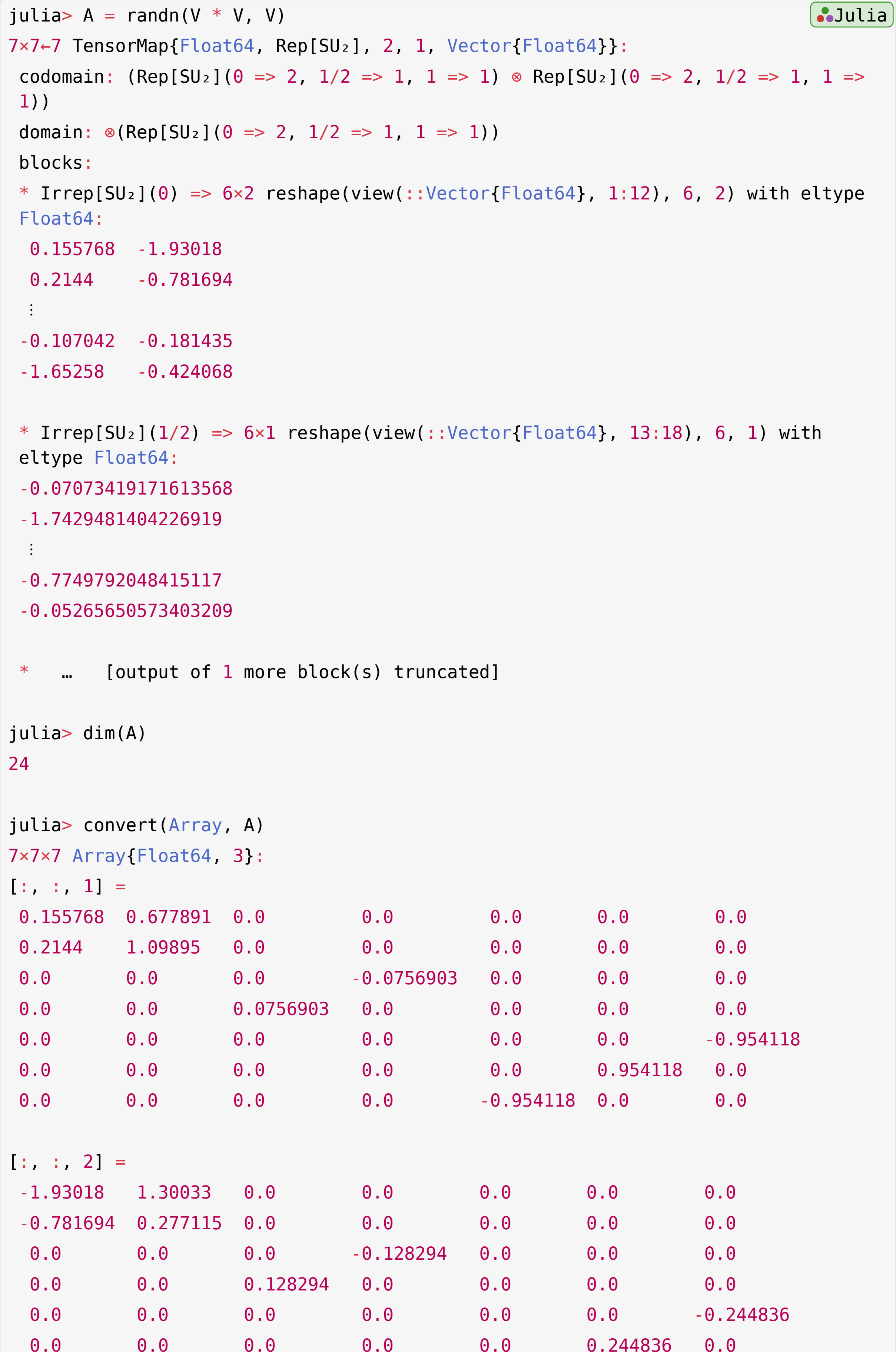
```julia
julia> for s in sectors(V)
           @show s, dim(V, s)
       end
(s, dim(V, s)) = (Irrep[U₁](0), 3)
(s, dim(V, s)) = (Irrep[U₁](1), 2)
(s, dim(V, s)) = (Irrep[U₁](-1), 1)

julia> U₁Space(-1 => 1, 0 => 3, 1 => 2) == GradedSpace(Irrep[U₁](1) => 2,
Irrep[U₁](0) => 3, Irrep[U₁](-1) => 1)
true

julia> supertype(GradedSpace)
ElementarySpace
```

Note that `GradedSpace` is not immediately parameterized by some group `G` , but actually by the set of irreducible representations of `G` , denoted as `Irrep[G]` . Indeed, `GradedSpace` also supports a grading that is derived from the fusion ring of a (unitary) pre-fusion category. Note furthermore that the order in which the charges and their corresponding subspace dimensionality are specified is irrelevant, and that the charges, henceforth called sectors (which is a more general name for charges or quantum numbers) are of a specific type, in this case `Irrep[U₁] == U1Irrep` . However, the `Vect[I]` constructor automatically converts the keys in the list of `Pair` s it receives to the correct type. Alternatively, we can directly create the sectors of the correct type and use the generic `GradedSpace` constructor. We can probe the subspace dimension of a certain sector `s` in a space `V` with `dim(V, s)` . Finally, note that `GradedSpace` still has the standard Euclidean inner product and we assume all representations to be unitary.

TensorKit.jl also allows for non-abelian symmetries such as `SU₂` . In this case, the vector space is characterized via the spin quantum number (i.e. the irrep label of `SU₂`) for each of its subspaces, and is created using `SU₂Space` (or `SU2Space` or `Rep[SU₂]` or `Vect[Irrep[SU₂]]`)

```julia
julia> V = SU₂Space(0 => 2, 1/2 => 1, 1 => 1)
Rep[SU₂](…) of dim 7:
   0 => 2
 1/2 => 1
   1 => 1

julia> dim(V)
7

julia> V == Vect[Irrep[SU₂]](0 => 2, 1 => 1, 1 // 2 => 1)
true
```

Note that now `V` has a two-dimensional subspace with spin zero, and two one-dimensional subspaces with spin 1/2 and spin 1. However, a subspace with spin `j` has an additional `2j + 1` dimensional degeneracy on which the irreducible representation acts. This brings the total dimension to `2*1 + 1*2 + 1*3` . Creating a tensor with `SU₂` symmetry yields



```julia
julia> A = randn(V * V, V)
7×7←7 TensorMap{Float64, Rep[SU₂], 2, 1, Vector{Float64}}:
 codomain: (Rep[SU₂](0 => 2, 1/2 => 1, 1 => 1) ⊗ Rep[SU₂](0 => 2, 1/2 => 1, 1 =>
 1))
 domain: ⊗(Rep[SU₂](0 => 2, 1/2 => 1, 1 => 1))
 blocks:
 * Irrep[SU₂](0) => 6×2 reshape(view(::Vector{Float64}, 1:12), 6, 2) with eltype
 Float64:
  0.155768  -1.93018
  0.2144    -0.781694
  ⋮
 -0.107042  -0.181435
 -1.65258   -0.424068

 * Irrep[SU₂](1/2) => 6×1 reshape(view(::Vector{Float64}, 13:18), 6, 1) with
 eltype Float64:
 -0.07073419171613568
 -1.7429481404226919
  ⋮
 -0.7749792048415117
 -0.05265650573403209

 *   …   [output of 1 more block(s) truncated]

julia> dim(A)
24

julia> convert(Array, A)
7×7×7 Array{Float64, 3}:
[:, :, 1] =
 0.155768  0.677891  0.0         0.0         0.0        0.0        0.0
 0.2144    1.09895   0.0         0.0         0.0        0.0        0.0
 0.0       0.0       0.0        -0.0756903   0.0        0.0        0.0
 0.0       0.0       0.0756903   0.0         0.0        0.0        0.0
 0.0       0.0       0.0         0.0         0.0        0.0       -0.954118
 0.0       0.0       0.0         0.0         0.0        0.954118   0.0
 0.0       0.0       0.0         0.0        -0.954118   0.0        0.0

[:, :, 2] =
 -1.93018   1.30033   0.0        0.0        0.0       0.0        0.0
 -0.781694  0.277115  0.0        0.0        0.0       0.0        0.0
  0.0       0.0       0.0       -0.128294   0.0       0.0        0.0
  0.0       0.0       0.128294   0.0        0.0       0.0        0.0
  0.0       0.0       0.0        0.0        0.0       0.0       -0.244836
  0.0       0.0       0.0        0.0        0.0       0.244836   0.0
```

```
  0.0       0.0       0.0        0.0       -0.244836  0.0        0.0

[:, :, 3] =
  0.0         0.0       -0.13168    0.0       0.0         0.0        0.0
  0.0         0.0        0.268879   0.0       0.0         0.0        0.0
 -0.0707342  -1.74295    0.0        0.0       0.0        -0.0304012  0.0
  0.0         0.0        0.0        0.0       0.0429939   0.0        0.0
  0.0         0.0        0.0       -0.632768  0.0         0.0        0.0
  0.0         0.0        0.447434   0.0       0.0         0.0        0.0
  0.0         0.0        0.0        0.0       0.0         0.0        0.0

[:, :, 4] =
  0.0         0.0       0.0       -0.13168   0.0  0.0         0.0
  0.0         0.0       0.0        0.268879  0.0  0.0         0.0
  0.0         0.0       0.0        0.0       0.0  0.0        -0.0429939
 -0.0707342  -1.74295  0.0        0.0       0.0  0.0304012   0.0
  0.0         0.0       0.0        0.0       0.0  0.0         0.0
  0.0         0.0       0.0       -0.447434  0.0  0.0         0.0
  0.0         0.0       0.632768   0.0       0.0  0.0         0.0

[:, :, 5] =
 0.0        0.0         0.0      0.0  0.395166   0.0       0.0
 0.0        0.0         0.0      0.0  0.384263   0.0       0.0
 0.0        0.0        -0.65231  0.0  0.0        0.0       0.0
 0.0        0.0         0.0      0.0  0.0        0.0       0.0
 0.178868  -0.0196716   0.0      0.0  0.0       -0.139685  0.0
 0.0        0.0         0.0      0.0  0.139685   0.0       0.0
 0.0        0.0         0.0      0.0  0.0        0.0       0.0

[:, :, 6] =
 0.0        0.0         0.0        0.0       0.0       0.395166   0.0
 0.0        0.0         0.0        0.0       0.0       0.384263   0.0
 0.0        0.0         0.0       -0.461253  0.0       0.0        0.0
 0.0        0.0        -0.461253   0.0       0.0       0.0        0.0
 0.0        0.0         0.0        0.0       0.0       0.0       -0.139685
 0.178868  -0.0196716   0.0        0.0       0.0       0.0        0.0
 0.0        0.0         0.0        0.0       0.139685  0.0        0.0

[:, :, 7] =
 0.0        0.0        0.0   0.0      0.0  0.0        0.395166
 0.0        0.0        0.0   0.0      0.0  0.0        0.384263
 0.0        0.0        0.0   0.0      0.0  0.0        0.0
 0.0        0.0        0.0  -0.65231  0.0  0.0        0.0
 0.0        0.0        0.0   0.0      0.0  0.0        0.0
 0.0        0.0        0.0   0.0      0.0  0.0       -0.139685
```

```
 0.178868  -0.0196716  0.0   0.0      0.0  0.139685   0.0

julia> norm(A) ≈ norm(convert(Array, A))
true
```

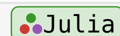
In this case, the full `Array` representation of the tensor has again many zeros, but it is less obvious to recognize the dense blocks, as there are additional zeros and the numbers in the original tensor data do not match with those in the `Array` . The reason is of course that the original tensor data now needs to be transformed with a construction known as fusion trees, which are made up out of the Clebsch-Gordan coefficients of the group. Indeed, note that the non-zero subblocks are also no longer labeled by a list of sectors, but by pairs of fusion trees. This will be explained further in the manual. However, the Clebsch-Gordan coefficients of the group are only needed to actually convert a tensor to an `Array` . For working with tensors with `SU₂Space` indices, e.g. contracting or factorizing them, the Clebsch-Gordan coefficients are never needed explicitly. Instead, recoupling relations are used to symbolically manipulate the basis of fusion trees, and this only requires what is known as the topological data of the group (or its representation theory).

In fact, this formalism extends beyond the case of group representations on vector spaces, and can also deal with super vector spaces (to describe fermions) and more general (unitary) fusion categories. Support for all of these generalizations is present in TensorKit.jl. Indeed, all of these concepts will be explained throughout the remainder of this manual, including several details regarding their implementation. However, to just use tensors and their manipulations (contractions, factorizations, ...) in higher level algorithms (e.g. tensoer network algorithms), one does not need to know or understand most of these details, and one can immediately refer to the general interface of the `TensorMap` type, discussed on the last page. Adhering to this interface should yield code and algorithms that are oblivious to the underlying symmetries and can thus work with arbitrary symmetric tensors.

Chapter 4

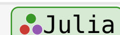
Vector spaces

From the Introduction, it should be clear that an important aspect in the definition of a tensor (map) is specifying the vector spaces and their structure in the domain and codomain of the map. The starting point is an abstract type `VectorSpace`



```julia
abstract type VectorSpace end
```

Technically speaking, this name does not capture the full generality that TensorKit.jl supports, as instances of subtypes of `VectorSpace` can encode general objects in linear monoidal categories, which are not necessarily vector spaces. However, in order not to make the remaining discussion too abstract or complicated, we will simply use the nomenclature of vector spaces. In particular, we define two abstract subtypes



```julia
abstract type ElementarySpace <: VectorSpace end
const IndexSpace = ElementarySpace

abstract type CompositeSpace{S<:ElementarySpace} <: VectorSpace end
```

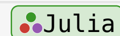
Here, `ElementarySpace` is a super type for all vector spaces (objects) that can be associated with the individual indices of a tensor, as hinted to by its alias `IndexSpace` .

On the other hand, subtypes of `CompositeSpace{S}` where `S <: ElementarySpace` are composed of a number of elementary spaces of type `S` . So far, there is a single concrete type `ProductSpace{S, N}` that represents the tensor product of `N` vector spaces of a homogeneous type `S` . Its properties are discussed in the section on Composite spaces, together with possible extensions for the future.

Throughout TensorKit.jl, the function `spacetype` returns the type of `ElementarySpace` associated with e.g. a composite space or a tensor. It works both on instances and in the type domain. Its use will be illustrated below.

4.1 Fields

Vector spaces (and linear categories more generally) are defined over a field of scalars $\mathbb{F}$. We define a type hierarchy to specify the scalar field, but so far only support real and complex numbers, via



```julia
abstract type Field end

struct RealNumbers <: Field end
struct ComplexNumbers <: Field end

const ℝ = RealNumbers()
const ℂ = ComplexNumbers()
```

Note that ℝ and ℂ can be typed as `\bbR` +TAB and `\bbC` +TAB. One reason for defining this new type hierarchy instead of recycling the types from Julia's `Number` hierarchy is to introduce some syntactic sugar without committing type piracy. In particular, we now have

```julia
julia> 3 ∈ ℝ
true

julia> 5.0 ∈ ℂ
true

julia> 5.0 + 1.0 * im ∈ ℝ
false

julia> Float64 ⊆ ℝ
true

julia> ComplexF64 ⊆ ℂ
true

julia> ℝ ⊆ ℂ
true

julia> ℂ ⊆ ℝ
false
```

and furthermore —probably more usefully— `ℝ^n` and `ℂ^n` create specific elementary vector spaces as described in the next section. The underlying field of a vector space or tensor `a` can be obtained with `field(a)` :

4.2 Elementary spaces

As mentioned at the beginning of this section, vector spaces that are associated with the individual indices of a tensor should be implemented as subtypes of `ElementarySpace` . As the domain and codomain of a tensor map will be the tensor product of such objects which all have the same type, it is important that associated vector spaces, such as the dual space, are objects of the same concrete type (i.e. with the same type parameters in case of a parametric type). In particular, every `ElementarySpace` should implement the following methods

`TensorKitSectors.dim` – Method.

```julia
dim(V::VectorSpace) -> Int
```

Return the total dimension of the vector space `V` as an Int.

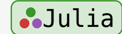
source

```julia
dim(P::ProductSpace{S, N}, s::NTuple{N, sectortype(S)}) where
{S<:ElementarySpace}
-> Int
```

Return the total degeneracy dimension corresponding to a tuple of sectors for each of the spaces in the tensor product, obtained as `prod(dims(P, s))` `.

source

`TensorKit.field` – Method.

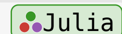


```julia
field(a) -> Type{𝔽 <: Field}
field(::Type{T}) -> Type{𝔽 <: Field}
```

Return the type of field over which object `a` (e.g. a vector space or a tensor) is defined. This also works in type domain.

source

`TensorKitSectors.dual` – Method.

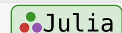


```julia
dual(V::VectorSpace) -> VectorSpace
```

Return the dual space of `V` ; also obtained via `V'` . This should satisfy `dual(dual(V)) == V` . It is assumed that `typeof(V) == typeof(V')` .

source

`Base.conj` – Method.



```julia
conj(V::S) where {S<:ElementarySpace} -> S
```

Return the conjugate space of `V` . This should satisfy `conj(conj(V)) == V` .

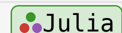
For `field(V)==ℝ` , `conj(V) == V` . It is assumed that `typeof(V) == typeof(conj(V))` .

source

For convenience, the dual of a space `V` can also be obtained as `V'` . Furthermore, it is sometimes necessary to test whether a space is a dual or conjugate space, for which the methods `isdual(::ElementarySpace)` and `isconj(::ElementarySpace)` should be implemented.

We furthermore define a trait type

`TensorKit.InnerProductStyle` – Type.



```julia
abstract type InnerProductStyle end
InnerProductStyle(V::VectorSpace) -> ::InnerProductStyle
InnerProductStyle(S::Type{<:VectorSpace}) -> ::InnerProductStyle
```

Trait to describe wether vector spaces exhibit an inner product structure, a.k.a. a unitary structure, which can take the following values:

- `EuclideanInnerProduct()` : the metric is the identity, making dual and conjugate spaces equivalent
- `NoInnerProduct()` : no metric and thus no relation between `dual(V)` or `conj(V)`

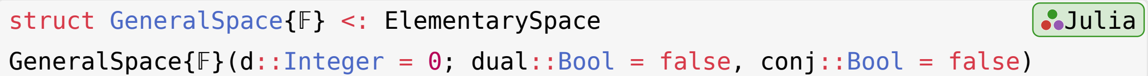
Furthermore, `EuclideanInnerProduct` is a subtype of `HasInnerProduct` , indicating that an inner product exists, and an isomorphism between the dual space and the conjugate space can be constructed. New inner product styles can be defined that subtype `HasInnerProduct` , for example to work with vector spaces with non-trivial metrics. However, at the moment TensorKit does not provide built-in support for such non-standard inner products.

source

to denote for a vector space `V` whether it has an inner product and thus a canonical mapping from `dual(V)` to `V` (for real fields `𝔽 ⊆ ℝ`) or from `dual(V)` to `conj(V)` (for complex fields). This mapping is provided by the metric, but no further support for working with vector spaces with general metrics is currently implemented.

A number of concrete elementary spaces are implemented in TensorKit.jl. There is concrete type `GeneralSpace` which is completely characterized by its field `𝔽`, its dimension and whether its the dual and/or complex conjugate of $\mathbb{F}^d$.

`TensorKit.GeneralSpace` – Type.



```julia
struct GeneralSpace{𝔽} <: ElementarySpace
GeneralSpace{𝔽}(d::Integer = 0; dual::Bool = false, conj::Bool = false)
```

A finite-dimensional space over an arbitrary field `𝔽` without additional structure. It is thus characterized by its dimension, and whether or not it is the dual and/or conjugate space. For a real field `𝔽`, the space and its conjugate are the same.

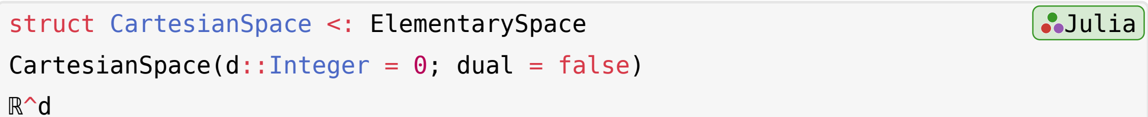
source

However, as the `InnerProductStyle` of `GeneralSpace` is currently set to `NoInnerProduct()`, this type of vector space is currently quite limited, though it supports constructing tensors and contracting them. However, most tensor factorizations will depend on the presence of an Euclidean inner product.

Spaces with the `EuclideanInnerProduct()` style, i.e. with a standard Euclidean metric, have the natural isomorphisms `dual(V) == V` (for `𝔽 == ℝ`) or `dual(V) == conj(V)` (for `𝔽 == ℂ`). In the language of the appendix on categories, this trait represents dagger or unitary categories, and these vector spaces support an `adjoint` operation.

In particular, two concrete types are provided:

`TensorKit.CartesianSpace` – Type.

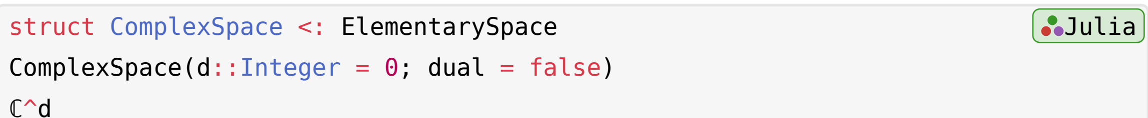


```julia
struct CartesianSpace <: ElementarySpace
CartesianSpace(d::Integer = 0; dual = false)
ℝ^d
```

A real Euclidean space $\mathbb{R}^d$. `CartesianSpace` has no additonal structure and is completely characterised by its dimension `d`. A `dual` keyword argument is accepted for compatibility with other space constructors, but is ignored since the dual of a Cartesian space is isomorphic to itself. This is the vector space that is implicitly assumed in most of matrix algebra.

source

`TensorKit.ComplexSpace` – Type.



```julia
struct ComplexSpace <: ElementarySpace
ComplexSpace(d::Integer = 0; dual = false)
ℂ^d
```

A standard complex vector space $\mathbb{C}^d$ with Euclidean inner product and no additional structure. It is completely characterised by its dimension and whether its the normal space or its dual (which is canonically isomorphic to the conjugate space).

source

They represent the Euclidean spaces $\mathbb{R}^d$ or C^d without further inner structure. They can be created using the syntax `CartesianSpace(d) == ℝ^d` and `ComplexSpace(d) == ℂ^d` , or `ComplexSpace(d, true) == ComplexSpace(d; dual = true) == (ℂ^d)'` for the dual space of the latter. Note that the brackets are required because of the precedence rules, since `d' == d` for `d::Integer` .

Some examples:

```julia
julia> dim(ℝ^10)
10

julia> (ℝ^10)' == ℝ^10
true

julia> isdual((ℂ^5))
false

julia> isdual((ℂ^5)')
true

julia> isdual((ℝ^5)')
false

julia> dual(ℂ^5) == (ℂ^5)' == conj(ℂ^5) == ComplexSpace(5; dual = true)
true

julia> field(ℂ^5)
ℂ

julia> field(ℝ^3)
ℝ

julia> typeof(ℝ^3)
CartesianSpace

julia> spacetype(ℝ^3)
CartesianSpace

julia> InnerProductStyle(ℝ^3)
EuclideanInnerProduct()

julia> InnerProductStyle(ℂ^5)
EuclideanInnerProduct()
```

Note

For `ℂ^n` the dual space is equal (or naturally isomorphic) to the conjugate space, but not to the space itself. This means that even for `ℂ^n` , arrows matter in the diagrammatic notation for categories or

for tensors, and in particular that a contraction between two tensor indices will check that one is living in the space and the other in the dual space. This is in contrast with several other software packages, especially in the context of tensor networks, where arrows are only introduced when discussing symmetries. We believe that our more puristic approach can be useful to detect errors (e.g. unintended contractions). Only with `ℝ^n` will their be no distinction between a space and its dual. When creating tensors with indices in `ℝ^n` that have complex data, a one-time warning will be printed, but most operations should continue to work nonetheless.

One more important concrete implementation of `ElementarySpace` with a `EuclideanInnerProduct()` is the `GradedSpace` type, which is used to represent a graded complex vector space, where the grading is provided by the irreducible representations of a group, or more generally, the simple objects of a unitary fusion category. We refer to the subsection on graded spaces on the next page for further information about `GradedSpace` .

4.3 Operations with elementary spaces

Instances of `ElementarySpace` support a number of useful operations. Firstly, we define the direct sum of two vector spaces `V1` and `V2` of the same `spacetype` (and with the same value of `isdual`) as `V1 ⊕ V2` , where `⊕` is obtained by typing `\oplus` +TAB. `zerospace(V)` corresponds to the identity or zero element with respect to this direct sum operation, i.e. it corresponds to a zero-dimensional space. Furthermore, `unitspace(V)` applied to an elementary space returns a one-dimensional space, that is isomorphic to the scalar field underlying the space itself. Finally, we have also introduced the non-standard convention `V1 ⊖ V2` (obtained by typing `\ominus` +TAB.) in order to obtain a space that is isomorphic to the quotient space of `V1` by `V2` , or thus, a particular choice of complement of `V2` in `V1` such that `V1 == V2 ⊕ (V1 ⊖ V2)` is satisfied.

Some examples illustrate this better.

```julia
julia> ℝ^5 ⊕ ℝ^3
ℝ^8

julia> ℂ^5 ⊕ ℂ^3
ℂ^8

julia> ℂ^5 ⊕ (ℂ^3)'
ERROR: SpaceMismatch: Direct sum of a vector space and its dual does not exist

julia> zerospace(ℂ^3)
ℂ^0

julia> unitspace(ℝ^3)
ℝ^1

julia> ℂ^5 ⊕ unitspace(ComplexSpace)
ℂ^6

julia> unitspace((ℂ^3)')
```

```
ℂ^1

julia> (ℂ^5) ⊕ unitspace((ℂ^5))
ℂ^6

julia> (ℂ^5)' ⊕ unitspace((ℂ^5)')
ERROR: SpaceMismatch: Direct sum of a vector space and its dual does not exist

julia> (ℝ^5) ⊖ (ℝ^3)
ℝ^2

julia> (ℂ^5) ⊖ (ℂ^3)
ℂ^2

julia> (ℂ^5)' ⊖ (ℂ^3)'
(ℂ^2)'

julia> ℂ^5 == ((ℂ^5) ⊖ (ℂ^3)) ⊕ (ℂ^3) == (ℂ^3) ⊕ ((ℂ^5) ⊖ (ℂ^3))
true
```

Note, finally, that we have defined `oplus` and `ominus` as ASCII alternatives for ⊕ and ⊖ respectively.

A second type of operation with elementary spaces is the function `flip(V::ElementarySpace)` , which returns a space that is isomorphic to `V` but has `isdual(flip(V)) == isdual(V')` , i.e., if `V` is a normal space, then `flip(V)` is a dual space. `flip(V)` is different from `dual(V)` in the case of `GradedSpace` . It is useful to flip a tensor index from a ket to a bra (or vice versa), by contracting that index with a unitary map from `V1` to `flip(V1)` .

While we provide some trivial examples here, we refer to the section on graded spaces for examples where `flip` acts non-trivially and produces results that are different than `dual` .

```julia
julia> flip(ℂ^4)
(ℂ^4)'

julia> flip(ℂ^4) ≅ ℂ^4
true

julia> flip(ℂ^4) == ℂ^4
false
```

Finally, we provide two methods `infimum(V1, V2)` and `supremum(V1, V2)` for elementary spaces `V1` and `V2` with the same `spacetype` and value of `isdual` . The former returns the "largest" elementary space `V::ElementarySpace` with the same value of `isdual` such that we can construct surjective morphisms from both `V1` and `V2` to `V` . Similarly, the latter returns the "smallest" elementary space `W::ElementarySpace` with the same value of `isdual` such that we can construct injective morphisms from both `V1` and `V2` to `W` . For `CartesianSpace` and `ComplexSpace` , this simply amounts to the space with minimal or maximal dimension, but it is again more interesting in the case of `GradedSpace` , as discussed on the next page. It is that case where `infimum(V1, V2)` might be different from either `V1` or `V2` , and similar for `supremum(V1, V2)` , which justifies the choice of these names over simply `min`

and `max`. Also note that these methods are a direct consequence of the partial order that we can define between vector spaces of the same `spacetype` more generally, as discussed below in the subsection More operations with vector spaces.

Some examples:

```julia
julia> infimum(ℝ^5, ℝ^3)
ℝ^3

julia> supremum(ℂ^5, ℂ^3)
ℂ^5

julia> supremum(ℂ^5, (ℂ^3)')
ERROR: SpaceMismatch: Supremum of space and dual space does not exist

julia> supremum((ℂ^5)', (ℂ^3)')
(ℂ^5)'
```

4.4 Composite spaces

Composite spaces are vector spaces that are built up out of individual elementary vector spaces of the same type. The most prominent (and currently only) example is a tensor product of `N` elementary spaces of the same type `S`:

`TensorKit.ProductSpace` – Type.

```julia
struct ProductSpace{S <: ElementarySpace, N} <: CompositeSpace{S}
ProductSpace(spaces::NTuple{N, S}) where {S <: ElementarySpace, N}
```

A `ProductSpace` is a tensor product space of `N` vector spaces of type `S <: ElementarySpace`. Only tensor products between `ElementarySpace` objects of the same type are allowed.

source

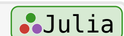
Given some `V1::S`, `V2::S`, `V3::S` of the same type `S<:ElementarySpace`, we can easily construct `ProductSpace{S, 3}((V1, V2, V3))` as `ProductSpace(V1, V2, V3)` or using `V1 ⊗ V2 ⊗ V3`, where `⊗` is simply obtained by typing `\otimes` +TAB. In fact, for convenience, also the regular multiplication operator `*` acts as tensor product between vector spaces, and as a consequence so does raising a vector space to a positive integer power, i.e.

```julia
julia> V1 = ℂ^2
ℂ^2

julia> V2 = ℂ^3
ℂ^3

julia> V1 ⊗ V2 ⊗ V1' == V1 * V2 * V1' == ProductSpace(V1, V2, V1') ==
ProductSpace(V1, V2) ⊗ V1'
true
```

```
julia> V1^3
(ℂ^2 ⊗ ℂ^2 ⊗ ℂ^2)

julia> dim(V1 ⊗ V2)
6

julia> dims(V1 ⊗ V2)
(2, 3)

julia> dual(V1 ⊗ V2 ⊗ V1')
(ℂ^2 ⊗ (ℂ^3)' ⊗ (ℂ^2)')

julia> spacetype(V1 ⊗ V2)
ComplexSpace

julia> spacetype(ProductSpace{ComplexSpace,3})
ComplexSpace
```

Here, the newly introduced function `dims` maps `dim` to the individual spaces in a `ProductSpace` and returns the result as a tuple. The rationale for the dual space of a `ProductSpace` being the tensor product of the dual spaces in reverse order is explained in the subsection on duality in the appendix on category theory.

Following Julia's Base library, the function `one` applied to an instance of `ProductSpace{S, N}` or of `S <: ElementarySpace` itself returns the multiplicative identity for these objects. Similar to Julia Base, `one` also works in the type domain. The multiplicative identity for vector spaces corresponds to the (monoidal) unit, which is represented as `ProductSpace{S, 0}(())` and simply printed as `one(S)` for the specific type `S` . Note, however, that `one(S)` is strictly speaking only the multiplicative identity when multiplied with `ProductSpace{S, N}` instances. For elementary spaces `V::S` , `V ⊗ one(V)` will yield `ProductSpace{S, 1}(V)` and not `V` itself. However, even though `V ⊗ one(V)` is not strictly equal to `V` , the object `ProductSpace(V)` , which can also be created as `⊗(V)` , does mathematically encapsulate the same vector space as `V` .



```
julia> one(V1)
one(ComplexSpace)

julia> typeof(one(V1))
ProductSpace{ComplexSpace, 0}

julia> V1 * one(V1) == ProductSpace(V1) == ⊗(V1)
true

julia> V1 * one(V1) == V1
false

julia> P = V1 * V2;
```

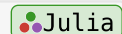
```
julia> one(P)
one(ComplexSpace)

julia> one(typeof(P))
one(ComplexSpace)

julia> P * one(P) == P == one(P) ⊗ P
true
```

In the future, other `CompositeSpace` types could be added. For example, the wave function of an N-particle quantum system in first quantization would require the introduction of a `SymmetricSpace{S, N}` or a `AntiSymmetricSpace{S, N}` for bosons or fermions respectively, which correspond to the symmetric (permutation invariant) or antisymmetric subspace of `V^N` , where `V::S` represents the Hilbert space of the single particle system. Other scientific fields, like general relativity, might also benefit from tensors living in subspace with certain symmetries under specific index permutations.

4.5 More operations with vector spaces

Vector spaces of the same `spacetype` can be given a partial order, based on whether there exist injective morphisms (a.k.a *monomorphisms*) or surjective morphisms (a.k.a. *epimorphisms*) between them. In particular, we define `ismonomorphic(V1, V2)` , with Unicode synonym `V1 ≾ V2` (obtained as `\precsim+TAB`), to express whether there exist monomorphisms in `V1 → V2` . Similarly, we define `isepimorphic(V1, V2)` , with Unicode synonym `V1 ≿ V2` (obtained as `\succsim+TAB`), to express whether there exist epimorphisms in `V1 → V2` . Finally, we define `isisomorphic(V1, V2)` , with Unicode alternative `V1 ≅ V2` (obtained as `\cong+TAB`), to express whether there exist isomorphism in `V1 → V2` . In particular `V1 ≅ V2` if and only if `V1 ≾ V2 && V1 ≿ V2` .

For completeness, we also export the strict comparison operators `≺` and `≻` (`\prec+TAB` and `\succ+TAB`), with definitions



```julia
≺(V1::VectorSpace, V2::VectorSpace) = V1 ≾ V2 && !(V1 ≿ V2)
≻(V1::VectorSpace, V2::VectorSpace) = V1 ≿ V2 && !(V1 ≾ V2)
```

However, as we expect these to be less commonly used, no ASCII alternative is provided.

In the context of `InnerProductStyle(V) <: EuclideanInnerProduct` , `V1 ≾ V2` implies that there exists isometries $W : V1 \to V2$ such that $W^\dagger \circ W = \mathrm{id}_{V1}$, while `V1 ≅ V2` implies that there exist unitaries $U : V1 \to V2$ such that $U^\dagger \circ U = \mathrm{id}_{V1}$ and $U \circ U^\dagger = \mathrm{id}_{V2}$.

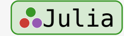
Note that spaces that are isomorphic are not necessarily equal. One can be a dual space, and the other a normal space, or one can be an instance of `ProductSpace` , while the other is an `ElementarySpace` . There will exist (infinitely) many isomorphisms between the corresponding spaces, but in general none of those will be canonical.

There are also a number of convenience functions to create isomorphic spaces. The function `fuse(V1, V2, ...)` or `fuse(V1 ⊗ V2 ⊗ ...)` returns an elementary space that is isomorphic to `V1 ⊗ V2 ⊗ ...` .

4.6 Space of morphisms

As mentioned in the introduction, we define tensor maps as linear maps from a `ProductSpace` domain to a `ProductSpace` codomain. The set of all tensor maps with a fixed domain and codomain constitutes a vector space, which we represent with the `HomSpace` type.

`TensorKit.HomSpace` – Type.



```julia
struct HomSpace{S<:ElementarySpace, P1<:CompositeSpace{S},
P2<:CompositeSpace{S}}
HomSpace(codomain::CompositeSpace{S}, domain::CompositeSpace{S}) where
{S<:ElementarySpace}
```

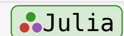
Represents the linear space of morphisms with codomain of type `P1` and domain of type `P2` . Note that `HomSpace` is not a subtype of `VectorSpace` , i.e. we restrict the latter to denote categories and their objects, and keep `HomSpace` distinct.

source

Aside from the standard constructor, a `HomSpace` instance can be created as either `domain → codomain` or `codomain ← domain` (where the arrows are obtained as `\to+TAB` or `\leftarrow+TAB` , and as `\rightarrow+TAB` respectively). The reason for first listing the codomain and than the domain will become clear in the section on tensor maps.

Note that `HomSpace` is not a subtype of `VectorSpace` , i.e. we restrict the latter to encode all spaces and generalizations thereof (i.e. objects in linear monoidal categories) that are associated with the indices and the domain and codomain of a tensor map. Even when these generalizations are no longer strictly vector spaces and have unconventional properties (such as non-integer dimensions), the space of tensor maps (homomorphisms) between a given domain and codomain, represented by a `HomSpace` instance, is always a vector space in the strict mathematical sense (with in particular an integer dimension). Because `HomSpace` and the different subtypes of `VectorSpace` represent very different mathematical concepts that do not directly interact, we have chosen to keep them separate in the type hierarchy.

Furthermore, on these `HomSpace` instances, we define a number of useful methods that are a precursor to the corresponding methods that we will define to manipulate the actual tensors, as illustrated in the following example:



```julia
julia> W = ℂ^2 ⊗ ℂ^3 → ℂ^3 ⊗ dual(ℂ^4)
(ℂ^3 ⊗ (ℂ^4)') ← (ℂ^2 ⊗ ℂ^3)

julia> field(W)
ℂ

julia> dual(W)
((ℂ^3)' ⊗ (ℂ^2)') ← (ℂ^4 ⊗ (ℂ^3)')

julia> adjoint(W)
(ℂ^2 ⊗ ℂ^3) ← (ℂ^3 ⊗ (ℂ^4)')

julia> spacetype(W)
ComplexSpace
```

```
julia> spacetype(typeof(W))
ComplexSpace

julia> W[1]
ℂ^3

julia> W[2]
(ℂ^4)'

julia> W[3]
(ℂ^2)'

julia> W[4]
(ℂ^3)'

julia> dim(W)
72

julia> domain(W)
(ℂ^2 ⊗ ℂ^3)

julia> codomain(W)
(ℂ^3 ⊗ (ℂ^4)')

julia> numin(W)
2

julia> numout(W)
2

julia> numind(W)
4

julia> numind(W) == numin(W) + numout(W)
true

julia> permute(W, ((2, 3), (1, 4)))
((ℂ^4)' ⊗ (ℂ^2)') ← ((ℂ^3)' ⊗ ℂ^3)

julia> flip(W, 3)
(ℂ^3 ⊗ (ℂ^4)') ← ((ℂ^2)' ⊗ ℂ^3)

julia> insertleftunit(W, 3)
(ℂ^3 ⊗ (ℂ^4)') ← (ℂ^1 ⊗ ℂ^2 ⊗ ℂ^3)
```

```
julia> insertrightunit(W, 2)
(ℂ^3 ⊗ (ℂ^4)' ⊗ ℂ^1) ← (ℂ^2 ⊗ ℂ^3)

julia> removeunit(insertrightunit(W, 2), 3)
(ℂ^3 ⊗ (ℂ^4)') ← (ℂ^2 ⊗ ℂ^3)

julia> TensorKit.compose(W, adjoint(W))
(ℂ^3 ⊗ (ℂ^4)') ← (ℂ^3 ⊗ (ℂ^4)')
```

Note that indexing `W` follows an order that first targets the spaces in the codomain, followed by the dual of the spaces in the domain. This particular convention is useful in combination with the instances of type `TensorMap` , which represent the actual morphisms living in such a `HomSpace` . Also note that `dim(W)` is here given by the product of the dimensions of the individual spaces, but that this is no longer true once symmetries are involved. At any time will `dim(::HomSpace)` represent the number of linearly independent morphisms in this space, or thus, the number of independent components that a corresponding `TensorMap` object will have.

A complete list of methods defined on `HomSpace` instances together with the corresponding documentation is provided in the library section on Vector spaces.

Chapter 5

Symmetries

5.1 Symmetries and symmetric tensors

When a physical system exhibits certain symmetries, it can often be described using tensors that transform covariantly with respect to the corresponding symmetry group, where this group acts as a tensor product of group actions on every tensor index separately. The group action on a single index, or thus, on the corresponding vector space, can be decomposed into irreducible representations (irreps). Here, we restrict to unitary representations, and thus assume that the corresponding vector spaces also have a natural Euclidean inner product. In particular, the Euclidean inner product between two vectors is invariant under the group action and thus transforms according to the trivial representation of the group.

The corresponding vector spaces will be canonically represented as $V = \bigoplus_a \mathbb{C}^{n_a} \otimes R_a$, where a labels the different irreps, n_a is the number of times irrep a appears and R_a is the vector space associated with irrep a. Irreps are also known as spin sectors (in the case of SU_2) or charge sectors (in the case of U_1), and we henceforth refer to a as a sector. The number of times n_a that sector a appears will be referred to as the degeneracy of sector a in the space V. In fact, the approach taken by TensorKit.jl goes beyond the case of irreps of groups, and, using the language from the Appendix on categories, sectors correspond to (equivalence classes of) simple objects in a unitary fusion or multifusion category, whereas the "representation spaces" V correspond to general (semisimple) objects in such a category. Nonetheless, many aspects of the construction of symmetric tensors can already be appreciated by considering the representation theory of a non-abelian group such as SU_2 or SU_3 as example. For practical reasons, we assume that there is a canonical order of the sectors, so that the vector space V is completely specified by the values of n_a.

When considering a tensor product of such representation spaces, they can again be decomposed into a direct sum of "coupled" sectors and associated degeneracy spaces. However, a non-trivial basis transformation is required to go from the tensor product basis to the basis of coupled sectors. The gain in efficiency (both in memory occupation and computation time) obtained from using symmetric (technically: equivariant) tensor maps is that, by Schur's lemma, they are block diagonal in the basis of coupled sectors. Hence, to exploit this block diagonal form, it is essential that we know the basis transformation from the individual (uncoupled) sectors appearing in the tensor product form of the domain and codomain, to the totally coupled sectors that label the different blocks. We refer to the latter as block sectors. The transformation from the uncoupled sectors in the domain (or codomain) of the tensor map to the block sector is encoded in a fusion tree (or splitting tree). Essentially, it is a sequential application of pairwise fusion as described by the group's Clebsch–Gordan (CG) coefficients. However, it turns out that we do not need to know or instantiate the actual CG coefficients that make up the fusion and splitting trees. Instead, we only need to know how the splitting and fusion trees transform under transformations such as interchanging the order of the incoming sectors or interchanging incoming and outgoing sectors. This information is known as the topological data of the group. It consists out of the fusion rules and the associativity relations encoded by the F-symbols, which are also known as recoupling coefficients or 6j-symbols (more accurately, the F-symbol is actually Racah's W-coefficients in the case of SU_2).

In the next three sections of the manual, we describe how the above concepts are implemented in TensorKit.jl in greater detail. Firstly, we describe how sectors and their associated topological data are encoded using a specialized interface and type hierarchy. The second section describes how to build spaces V composed of a direct sum of different sectors of the same type, and which operations are supported on those spaces. In the third section, we explain the details of constructing and manipulating fusion trees. Finally, we elaborate on the case of general fusion categories and the possibility of having fermionic or anyonic twists.

But first, on the remainder of this page, we provide a concise theoretical summary of the required data of the representation theory of a group. We refer to the appendix on categories, and in particular the subsection on topological data of a unitary fusion category, for more details.

Note

The infrastructure for defining sectors is actually implemented in a standalone package, TensorKitSectors.jl, that is imported and reexported by TensorKit.jl.

Note

On this and the next page of the manual, we assume some familiarity with the representation theory of non-abelian groups, and the structure of a symmetric tensor. For a more pedagogical introduction based on physical examples, we recommend reading the first appendix, which provides a tutorial-style introduction on the construction of symmetric tensors.

5.2 Representation theory and unitary fusion categories

Let the different irreps or sectors be labeled as a, b, c, … First and foremost, we need to specify the *fusion rules* $a \otimes b = \bigoplus N_c^{ab} c$ with N_c^{ab} some non-negative integers. The meaning of the fusion rules is that the space of covariant maps $R_a \otimes R_b \to R_c$ (or vice versa $R_c \to R_a \otimes R_b$) has dimension N_c^{ab}. In particular, there should always exists a unique trivial sector u (called the identity object I or 1 in the language of categories) such that $a \otimes u = a = u \otimes a$ for every other sector a. Furthermore, with respect to every sector a there should exist a unique sector $\bar{a}$ such that $N_u^{a\bar{a}} = 1$, whereas for all $b \neq \bar{a}$, $N_u^{ab} = 0$. For irreps of groups, $\bar{a}$ corresponds to the complex conjugate of the representation a, or some representation isomorphic to it. For example, for the representations of SU_2, the trivial sector corresponds to spin zero and all irreps are self-dual (i.e. $a = \bar{a}$), meaning that the conjugate representation is isomorphic to the non-conjugated one (they are however not equal but related by a similarity transform).

In particular, we now assume the existence of a basis for the N_c^{ab}-dimensional space of covariant maps $R_c \to R_a \otimes R_b$, which consists of unitary tensor maps $X_{c,\mu}^{ab} : R_c \to R_a \otimes R_b$ with $\mu = 1, …, N_c^{ab}$ such that

$$X_{c,\mu}^{ab})^\dagger X_{c,\nu}^{ab} = \delta_{\mu,\nu} \mathrm{id}_{R_c}$$

and

$$\sum_c \sum_{\mu=1}^{N_c^{ab}} X_{c,\mu}^{ab} (X_{c,\mu}^{ab})^\dagger = \mathrm{id}_{R_a \otimes R_b}$$

The tensors $X^{ab}_{c,\mu}$ are the splitting tensors, and because we restrict to unitary representations (or unitary categories), the corresponding fusion tensors are obtained by hermitian conjugation. Different choices of orthonormal bases would be related by a unitary basis transform within the space, i.e. acting on the multiplicity label $\mu = 1, ..., N^{ab}_c$. For SU_2, where N^{ab}_c is zero or one and the multiplicity labels are absent, this freedom reduces to a phase factor. In a standard convention, the entries of $X^{ab}_{c,\mu}$ are precisely given by the CG coefficients. However, the point is that we do not need to know the tensors $X^{ab}_{c,\mu}$ explicitly, but only the topological data of (the representation category of) the group, which describes the following transformation:

- F-move or recoupling: the transformation from $(a \otimes b) \otimes c$ to $a \otimes (b \otimes c)$:

$$(X^{ab}_{e,\mu} \otimes \mathrm{id}_c) \circ X^{ec}_{d,\nu} = \sum_{f,\kappa,\lambda} [F^{abc}_d]^{f,\kappa\lambda}_{e,\mu\nu} (\mathrm{id}_a \otimes X^{bc}_{f,\kappa}) \circ X^{af}_{d,\lambda}$$

- Braiding or permuting: the transformation from $a \otimes b$ to $b \otimes a$ as defined by $\tau_{a,b} : R_a \otimes R_b \to R_b \otimes R_a$:

$$\tau_{R_a,R_b} \circ X^{ab}_{c,\mu} = \sum_{\nu} [R^{ab}_c]^{\nu}_{\mu} X^{ba}_{c,\nu}$$

The dimensions of the spaces R_a on which representation a acts are denoted as d_a and referred to as quantum dimensions. In particular $d_u = 1$ and $d_a = d_{\bar{a}}$. This information is also encoded in the F-symbol as $d_a = |\ [F^{a\bar{a}a}_a]^u_u\ |^{-1}$. Note that there are no multiplicity labels in that particular F-symbol as $N^{a\bar{a}}_u = 1$.

There is a graphical representation associated with the fusion tensors and their manipulations, which we summarize here:

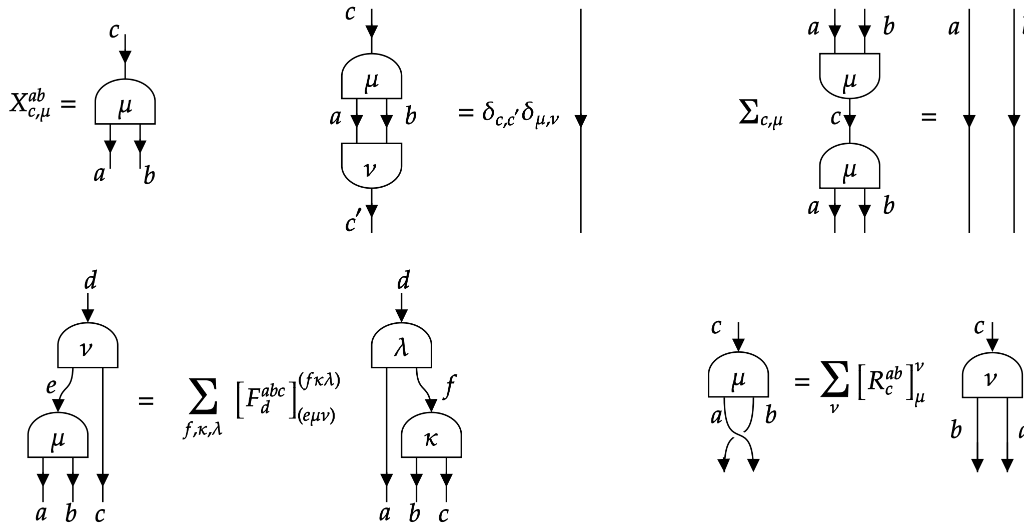


Figure 5.1: summary

We refer to the appendix on category theory, and in particular the section on topological data of a unitary fusion category for further details.

Finally, for the implementation, it will be useful to distinguish between a number of different possibilities regarding the fusion rules. If, for every a and b, there is a unique c such that $a \otimes b = c$ (i.e. $N^{ab}_c = 1$ and $N^{ab}_{c'} = 0$ for all other c'), the sector type is said to have unique fusion. The representations of a group have this property if and only if the group multiplication law is commutative, i.e. if the group is abelian. In that case, all spaces R_a associated with the representation are one-dimensional and thus

trivial. In the case of representations of non-abelian groups, or in the more general categorical case, there will always be at least one pair of sectors a and b (not necessarily distinct) for which the fusion product $a \otimes b$ contains more than one sector c with non-zero N_c^{ab}. In those cases, we find it useful to further distinguish between sector types for which N_c^{ab} only takes the values zero or one, such that no multiplicity labels (the Greek letters μ, ... are needed), e.g. the representations of SU_2, and those where some N_c^{ab} are larger than one, e.g. the representations of SU_3.

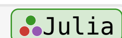
Chapter 6

Sectors

The first ingredient in order to define and construct symmetric tensors, is a framework to define symmetry sectors and their assocated fusion rules and topological data. TensorKitSectors.jl defines an abstract supertype `Sector` that all sectors will be subtypes of

`TensorKitSectors.Sector` – Type.

```julia
abstract type Sector
```



Abstract type for representing the (isomorphism classes of) simple objects in (unitary and pivotal) (pre-)fusion categories, e.g. the irreducible representations of a finite or compact group. Subtypes `I <: Sector` as the set of labels of a `GradedSpace` .

Every new `I <: Sector` should implement the following methods:

- `unit(::Type{I})` : unit element of `I` . If there are multiple, implement `allunits(::Type{I})` instead.
- `dual(a::I)` : $\bar{a}$, conjugate or dual label of a
- `⊗(a::I, b::I)` : iterable with unique fusion outputs of $a \otimes b$ (i.e. don't repeat in case of multiplicities)
- `Nsymbol(a::I, b::I, c::I)` : number of times `c` appears in `a ⊗ b` , i.e. the multiplicity
- `FusionStyle(::Type{I})` : `UniqueFusion()` , `SimpleFusion()` or `GenericFusion()`
- `BraidingStyle(::Type{I})` : `Bosonic()` , `Fermionic()` , `Anyonic()` , …
- `Fsymbol(a::I, b::I, c::I, d::I, e::I, f::I)` : F-symbol: scalar (in case of `UniqueFusion` / `SimpleFusion`) or rank-4 array (in case of `GenericFusion`)
- `Rsymbol(a::I, b::I, c::I)` : R-symbol: scalar (in case of `UniqueFusion` / `SimpleFusion`) or matrix (in case of `GenericFusion`)
- `isless(a::I, b::I)` : defines a canonical ordering of sectors
- `hash(a::I)` : hash function for sectors

and optionally

- `dim(a::I)` : quantum dimension of sector `a`
- `frobenius_schur_indicator(a::I)` : Frobenius-Schur indicator of `a` (1, 0, −1)
- `frobenius_schur_phase(a::I)` : Frobenius-Schur phase of `a` (±1)
- `sectorscalartype(::Type{I})` : scalar type of F- and R-symbols
- `Bsymbol(a::I, b::I, c::I)` : B-symbol: scalar (in case of `UniqueFusion` / `SimpleFusion`) or matrix (in case of `GenericFusion`)
- `twist(a::I)` -> twist of sector `a`

Furthermore, `iterate` and `Base.IteratorSize` should be made to work for the singleton type `SectorValues{I}` .

To help with the implementation of `⊗(a::I, b::I)` as an iterator, the provided `struct` type `SectorProductIterator{I}` can be used, which stores `a` and `b` and requires the implementation of `Base.iterate(::SectorProductIterator{I}, state...)` .

source

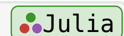
Any concrete subtype of `Sector` should be such that its instances represent a consistent set of sectors, corresponding to the irreps of some group, or, more generally, the simple objects of a (unitary) fusion category. Throughout TensorKit.jl, the method `sectortype` can be used to query the subtype of `Sector` associated with a particular object, i.e. a vector space, fusion tree, tensor map, or a sector. It works on both instances and in the type domain, and its use will be illustrated further on.

6.1 Minimal sector interface

The minimal data to completely specify a type of sector closely matches the topological data of a fusion category as reviewed in the appendix on category theory, and is given by:

- The fusion rules, i.e. $a \otimes b = \bigoplus N_c^{ab} c$, implemented as the function `Nsymbol(a, b, c)` .
- The list of fusion outputs from $a \otimes b$ while this information is contained in N_c^{ab}, it might be costly or impossible to iterate over all possible values of `c` and test `Nsymbol(a,b,c)` ; instead we require for `a ⊗ b` , or equivalently, `otimes(a, b)` , to return an iterable object (e.g. tuple or array, but see below for a dedicated iterator struct) that generates all *unique* `c` for which $N_c^{ab} \neq 0$ (so only once for all c with $N_c^{ab} \geq 1$).
- The identity object `u` , such that $a \otimes u = a = u \otimes a$, implemented as the function `unit(a)` (and also in type domain), but `one(a)` from Julia Base also works as an alias to `unit(a)` .
- The dual or conjugate object $\overline{a}$ for which $N_u^{a\bar{a}} = 1$, implemented as the function `dual(a)` . Because we restrict to unitary categories, `conj(a)` from the Julia `Base` library is also defined as an alias to `dual(a)` .
- The F-symbol or recoupling coefficients $[F_d^{abc}]_e^f$ implemented as the function `Fsymbol(a, b, c, d, e, f)` .
- If the category is braided (see below), the R-symbol R_c^{ab} implemented as the function `Rsymbol(a, b, c)` .

Furthermore, sectors should provide information about the structure of their fusion rules. For irreps of Abelian groups, we have that for every a and b, there exists a unique c such that $a \otimes b = c$, i. . there is only a single fusion channel. This follows simply from the fact that all irreps are one-dimensional. In all other cases, there is at least one pair of (a, b) exists such that $a \otimes b$ has multiple fusion outputs. This is often referred to as non-abelian fusion, and is the case for the irreps of a non-abelian group or some more general fusion category. We however still distinguish between the case where all entries of $N_c^{ab} \leqq 1$, i.e. they are zero or one. In that case, $[F_d^{abc}]_e^f$ and R_c^{ab} are scalars. If some $N_c^{ab} > 1$, it means that the same sector c can appear more than once in the fusion product of a and b, and we need to introduce some multiplicity label μ for the different copies, and $[F_d^{abc}]_e^f$ and R_c^{ab} are respectively four- and two-dimensional arrays labelled by these multiplicity indices. To encode these different possibilities, we define a Holy-trait called `FusionStyle` , i.e. a type hierarchy



```
abstract type FusionStyle end
struct UniqueFusion <: FusionStyle end # unique fusion output when fusing two sectors
abstract type MultipleFusion <: FusionStyle end
```

```
struct SimpleFusion <: MultipleFusion end # multiple fusion but multiplicity free
struct GenericFusion <: MultipleFusion end # multiple fusion with multiplicities
const MultiplicityFreeFusion = Union{UniqueFusion, SimpleFusion}
```

New sector types `I <: Sector` should then indicate which fusion style they have by defining `FusionStyle(::Type{I})` .

In a similar manner, it is useful to distinguish between the structure and the different styles of the braiding of a sector type. Remember that for group representations, braiding acts as swapping or permuting the vector spaces involved. By definition, applying this operation twice leads us back to the original situation. If that is the case, the braiding is said to be symmetric. For more general fusion categories, associated with the physics of anyonic particles, this is generally not the case. Some categories do not even support a braiding rule, as this requires at least that $a \otimes b$ and $b \otimes a$ have the same fusion outputs for every a and b. When braiding is possible, it might not be symmetric, and as a result, permutations of tensor indices are not unambiguously defined. The correct description is in terms of the braid group. This will be discussed in more detail below. Fermions are somewhat in between, as their braiding is symmetric, but they have a non-trivial *twist*. We thereto define a new trait `BraidingStyle` with associated the type hierarchy

```julia
abstract type HasBraiding <: BraidingStyle end
struct NoBraiding <: BraidingStyle end
abstract type SymmetricBraiding <: HasBraiding end # symmetric braiding => actions of permutation group are well defined
struct Bosonic <: SymmetricBraiding end # all twists are one
struct Fermionic <: SymmetricBraiding end # twists one and minus one
struct Anyonic <: HasBraiding end
```

New sector types `I <: Sector` should then indicate which fusion style they have by defining `BraidingStyle(::Type{I})` .

Note that `Bosonic()` braiding does not mean that all permutations are trivial and $R^{ab}_c = 1$, but that $R^{ab}_c R^{ba}_c = 1$. For example, for the irreps of SU_2, the R-symbol associated with the fusion of two spin-1/2 particles to spin zero is -1, i.e. the singlet of two spin-1/2 particles is antisymmetric under swapping the two constituents. For a `Bosonic()` braiding style, all twists are simply $+1$. The case of fermions and anyons are discussed below.

For practical reasons, we also require some additional methods to be defined:

- `hash(a, h)` creates a hash of sectors, because sectors and objects created from them are used as keys in lookup tables (i.e. dictionaries). Julia provides a default implementation of `hash` for every new type, but it can be useful to overload it for efficiency, or to ensure that the same hash is obtained for different instances that represent the same sector (e.g. when the sector type is not a bitstype).
- `isless(a, b)` associates a canonical order to sectors (of the same type), in order to unambiguously represent representation spaces $V = \bigoplus_a \mathbb{C}^{n_a} \otimes R_a$.

Lastly, we sometimes need to iterate over different values of a sector type `I <: Sector` , or at least have some basic information about the number of possible values of `I` Hereto, TensorKitSectors.jl defines `Base.values(I::Type{<:Sector})` to return the singleton instance of the parametric type `SectorValues{I}` , which should behave as an iterator over all possible values of the sector type `I` . This means the following methods should be implemented for a new sector type `I <: Sector` :

- `Base.iterate(::Type{SectorValues{I}} [, state])` should implement the iterator interface so as to enable iterating over all values of the sector `I` according to the canonical order defined by `isless` .
- `Base.IteratorSize(::Type{SectorValues{I}})` should return `HasLength()` if the number of different values of sector `I` is finite and rather small, and `SizeUnknown()` or `IsInfinite()` otherwise. This is used to encode the degeneracies of the different sectors in a `GradedSpace` object efficiently, as discussed in the next section on Graded spaces.
- If `IteratorSize(::Type{SectorValues{I}}) == HasLength()` , then `Base.length(::Type{SectorValues{I}})` should return the number of different values of sector `I` .

Furthermore, the standard definitions `Base.IteratorEltype(::Type{SectorValues{I}}) = HasEltype()` and `Base.eltype(::Type{SectorValues{I}}) = I` are provided by default in TensorKitSectors.jl.

Note

A recent update in TensorKitSectors.jl has extended the minimal interface to also support multi-fusion categories, for which in particular the unit object is non-simple. We do not discuss this extension here, but refer to the documentation of `UnitStyle` , `leftunit` , `rightunit` and `allunits` for more details.

6.2 Additional methods

The sector interface contains a number of additional methods, that are useful, but whose return value can be computed from the minimal interface defined in the previous subsection. However, new sector types can override these default fallbacks with more efficient implementations.

Firstly, the canonical order of sectors allows to enumerate the different values, and thus to associate each value with an integer. Hereto, the following methods are defined:

- `Base.getindex(::SectorValues{I}, i::Int)` : returns the sector instance of type `I` that is associated with integer `i` . The fallback implementation simply iterates through `values(I)` up to the `i` th value.
- `findindex(::SectorValues{I}, c::I)` : reverse mapping that associates an index `i::Integer ∈ 1:length(values(I))` to a given sector `c::I` . The fallback implementation simply searches linearly through the `values(I)` iterator.

Note that `findindex` acts similar to `Base.indexin` , but with the order of the arguments reversed (so that is more similar to `getindex`), and returns an `Int` rather than an `Array{0, Union{Int, Nothing}}` .

Secondly, it is often useful to know the scalar type in which the topological data in the F- and R-symbols are expressed. For this, the method `sectorscalartype(I::Type{<:Sector})` is provided, which has a default implementation that uses type inference on the return values of `Fsymbol` and `Rsymbol` . This function is also used to define `Base.isreal(I::Type{<:Sector})` , which indicates whether all topological data are real numbers. This is important because, if complex numbers appear in the topological data, it means tensor data will necessarily become complex after simple manipulations such as permuting indices, and should therefore probably be stored as complex numbers from the start.

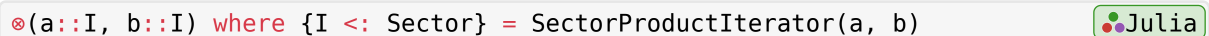
Finally, additional topological data can be extracted from the minimal interface. In particular, the quantum dimensions d_a and Frobenius-Schur phase χ_a and indicator (only if $a == \overline{a}$)

are encoded in the F-symbol. They are obtained as `dim(a)` , `frobenius_schur_phase(a)` and `frobenius_schur_indicator(a)` . These functions have default definitions which compute the requested data from `Fsymbol(a, conj(a), a, a, unit(a), unit(a))` , but they can be overloaded in case the value can be computed more efficiently. The same holds for related fusion manipulations such as the B-symbol, which is obtained as `Bsymbol(a, b, c)` . Finally, the twist associated with a sector `a` is obtained as `twist(a)` , which also has a default implementation in terms of the R-symbol. In addition, the function `isunit` is provided to facilitate checking whether a sector is a unit sector, in particular for the non-trivial case of the multi-fusion category case, which we do not discuss here.

6.3 Additional tools

The fusion product `a ⊗ b` of two sectors `a` and `b` is required to return an iterable object that generates all unique fusion outputs `c` for which $N_c^{ab} \geq 0$. When this list can easily be computed or constructed, it can be returned as a tuple or an array. However, when taking type stability and (memory) efficiency into account, it is often preferable to return a lazy iterator object that generates the different fusion outputs on the fly. Indeed, a tuple result is only type stable when the number of fusion outputs is constant for all possible inputs `a` and `b` , whereas a `Vector` result requires heap allocation.

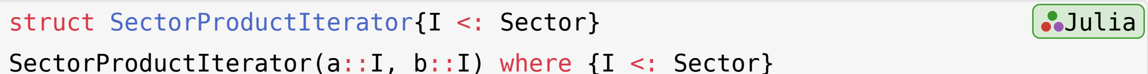
By default, TensorKitSectors.jl defines



```julia
⊗(a::I, b::I) where {I <: Sector} = SectorProductIterator(a, b)
```

where `TensorKitSectors.SectorProductIterator` is defined as

`TensorKitSectors.SectorProductIterator` – Type.



```julia
struct SectorProductIterator{I <: Sector}
SectorProductIterator(a::I, b::I) where {I <: Sector}
```

Custom iterator to represent the (unique) fusion outputs of $a \otimes b$.

Custom sectors that aim to use this have to provide the following functionality:

- `Base.iterate(::SectorProductIterator{I}, state...) where {I <: Sector}` : iterate over the fusion outputs of `a ⊗ b`

If desired and it is possible to easily compute the number of unique fusion outputs, it is also possible to define `Base.IteratorSize(::Type{SectorProductIterator{I}}) = Base.HasLength()` , in which case `Base.length(::SectorProductIterator{I})` has to be implemented.

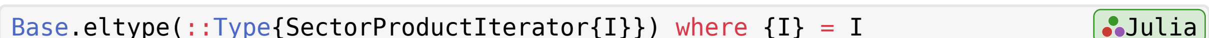
See also `⊗` .

source

and can serve as a general iterator type. For defining the fusion rules of a sector `I` , instead of implementing `⊗(::I, ::I)` directly, it is thus possible to instead implement the iterator interface for `SectorProductIterator{I}` , i.e. provide definitions for

- `Base.iterate(::SectorProductIterator{I}[, state])`
- `Base.IteratorSize(::Type{SectorProductIterator{I}})`
- `Base.length(::SectorProductIterator{I})` (if applicable)

TensorKitSectors.jl already defines



```julia
Base.eltype(::Type{SectorProductIterator{I}}) where {I} = I
```

and sets `Base.IteratorEltype(::Type{SectorProductIterator{I}})` accordingly. Furthermore, it provides custom pretty printing, so that `SectorProductIterator{I}(a, b)` is displayed as `a ⊗ b`.

6.4 Group representations

In this subsection, we give an overview of some existing sector types provided by TensorKitSectors.jl. We also discuss the implementation of some of them in more detail, in order to illustrate the interface defined above.

The first sector type is called `Trivial`, and corresponds to the case where there is actually no symmetry, or thus, the symmetry is the trivial group with only an identity operation and a trivial representation. Its representation theory is particularly simple:

```julia
struct Trivial <: Sector end

# basic properties
unit(::Type{Trivial}) = Trivial()
dual(::Trivial) = Trivial()
Base.isless(::Trivial, ::Trivial) = false

# fusion rules
⊗(::Trivial, ::Trivial) = (Trivial(),)
Nsymbol(::Trivial, ::Trivial, ::Trivial) = true
FusionStyle(::Type{Trivial}) = UniqueFusion()
Fsymbol(::Trivial, ::Trivial, ::Trivial, ::Trivial, ::Trivial, ::Trivial) = 1

# braiding rules
Rsymbol(::Trivial, ::Trivial, ::Trivial) = 1
BraidingStyle(::Type{Trivial}) = Bosonic()

# values iterator
Base.IteratorSize(::Type{SectorValues{Trivial}}) = HasLength()
Base.length(::SectorValues{Trivial}) = 1
Base.iterate(::SectorValues{Trivial}, i = false) = return i ? nothing :
(Trivial(), true)
function Base.getindex(::SectorValues{Trivial}, i::Int)
    return i == 1 ? Trivial() : throw(BoundsError(values(Trivial), i))
end
findindex(::SectorValues{Trivial}, c::Trivial) = 1
```

The `Trivial` sector type is special cased in the construction of tensors, so that most of these definitions are not actually used.

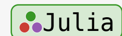
The most important class of sectors are irreducible representations of groups. As we often use the group itself as a type parameter, an associated type hierarchy for groups has been defined, namely

```julia
abstract type Group end
abstract type AbelianGroup <: Group end
```

```
abstract type Cyclic{N} <: AbelianGroup end
abstract type Dihedral{N} <: Group end
abstract type U₁ <: AbelianGroup end
abstract type CU₁ <: Group end

const ℤ{N} = Cyclic{N}
const ℤ₂ = ℤ{2}
const ℤ₃ = ℤ{3}
const ℤ₄ = ℤ{4}
const D₃ = Dihedral{3}
const D₄ = Dihedral{4}
const SU₂ = SU{2}
```

Groups themselves are abstract types without any functionality (at least for now). However, as will become clear instantly, it is useful to identify abelian groups, because their representation theory is particularly simple. We also provide a number of convenient Unicode aliases. These group names are probably self-explanatory, except for `CU₁` which is explained below.

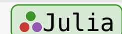
Irreps of groups will then be defined as subtypes of the abstract type



```
abstract type AbstractIrrep{G<:Group} <: Sector end # irreps have integer quantum dimensions
BraidingStyle(::Type{<:AbstractIrrep}) = Bosonic()
```

We will need different data structures to represent irreps of different groups, but it would be convenient to easily obtain the relevant structure for a given group `G` in a uniform manner. Hereto, we define a singleton type `IrrepTable` with an associated exported constant `Irrep = IrrepTable()` as the only instance. When a concrete type for representing the the irreps of a certain group `G` is iplemented, this type can the be "discovered" or obtained as `Irrep[G]` , provided it was registered by defining `Base.getindex(::IrrepTable, ::Type{G})` to return the concrete type.

Furthermore, we combine the more common functionality for irreps of abelian groups



```
const AbelianIrrep{G} = AbstractIrrep{G} where {G <: AbelianGroup}
FusionStyle(::Type{<:AbelianIrrep}) = UniqueFusion()
Base.sectorscalartype(::Type{<:AbelianIrrep}) = Int

Nsymbol(a::I, b::I, c::I) where {I <: AbelianIrrep} = c == first(a ⊗ b)
function Fsymbol(a::I, b::I, c::I, d::I, e::I, f::I) where {I <: AbelianIrrep}
    return Int(Nsymbol(a, b, e) * Nsymbol(e, c, d) * Nsymbol(b, c, f) * Nsymbol(a,
    f, d))
end
frobenius_schur_phase(a::AbelianIrrep) = 1
Asymbol(a::I, b::I, c::I) where {I <: AbelianIrrep} = Int(Nsymbol(a, b, c))
Bsymbol(a::I, b::I, c::I) where {I <: AbelianIrrep} = Int(Nsymbol(a, b, c))
Rsymbol(a::I, b::I, c::I) where {I <: AbelianIrrep} = Int(Nsymbol(a, b, c))
```

With these common definition in place, we implement the representation theory of the most common Abelian groups, starting with U_1, the full implementation of which is given by

```julia
struct U1Irrep <: AbstractIrrep{U₁}
    charge::HalfInt
end
Base.getindex(::IrrepTable, ::Type{U₁}) = U1Irrep
Base.convert(::Type{U1Irrep}, c::Real) = U1Irrep(c)

# basic properties
charge(c::U1Irrep) = c.charge
unit(::Type{U1Irrep}) = U1Irrep(0)
dual(c::U1Irrep) = U1Irrep(-charge(c))
@inline function Base.isless(c1::U1Irrep, c2::U1Irrep)
    return isless(abs(charge(c1)), abs(charge(c2))) || zero(HalfInt) < charge(c1)
    == -charge(c2)
end

# fusion rules
⊗(c1::U1Irrep, c2::U1Irrep) = (U1Irrep(charge(c1) + charge(c2)),)

# values iterator
Base.IteratorSize(::Type{SectorValues{U1Irrep}}) = IsInfinite()
function Base.iterate(::SectorValues{U1Irrep}, i::Int = 0)
    return i <= 0 ? (U1Irrep(half(i)), (-i + 1)) : (U1Irrep(half(i)), -i)
end
function Base.getindex(::SectorValues{U1Irrep}, i::Int)
    i < 1 && throw(BoundsError(values(U1Irrep), i))
    return U1Irrep(iseven(i) ? half(i >> 1) : -half(i >> 1))
end
function findindex(::SectorValues{U1Irrep}, c::U1Irrep)
    return (n = twice(charge(c)); 2 * abs(n) + (n <= 0))
end

# hashing
Base.hash(c::U1Irrep, h::UInt) = hash(c.charge, h)
```

A few comments are in order: The `getindex` definition just below the type definition provides the mechanism to obtain `U1Irrep` as `Irrep[U₁]` , as discussed above. The `Base.convert` definition, while not required by the minimal sector interface, allows to convert real numbers to the corresponding type of sector, and thus to omit the type information of the sector whenever this is clear from the context. The `charge` function is again not part of the minimal sector interface, and is specific to `U1Irrep` (and `ZNIrrep` discussed next), as a mere convenience function to access the charge value. Finally, in the definition of `U1Irrep` , `HalfInt <: Number` is a Julia type defined in HalfIntegers.jl, which is also used for `SU2Irrep` below, that stores integer or half integer numbers using twice their value. Strictly speaking, the linear representations of `U₁` can only have integer charges, and fractional charges lead to a projective representation. It can be useful to allow half integers in order to describe spin 1/2 systems

with an axis rotation symmetry. As a user, you should not worry about the details of `HalfInt` and additional methods for automatic conversion and pretty printing are provided, as illustrated by the following example

```julia
julia> Irrep[U₁](0.5)
Irrep[U₁](1/2)

julia> U1Irrep(0.4)
ERROR: InexactError: Int64(0.8)

julia> U1Irrep(1) ⊗ Irrep[U₁](1//2)
(Irrep[U₁](3/2),)

julia> u = first(U1Irrep(1) ⊗ Irrep[U₁](1//2))
Irrep[U₁](3/2)

julia> Nsymbol(u, dual(u), unit(u))
true
```

We similarly implement the irreps of the finite cyclic groups $\mathbb{Z}_N$, where we distinguish between small and large values of `N` to optimize storage. The implementation is given by

```julia
const SMALL_ZN_CUTOFF = (typemax(UInt8) + 1) ÷ 2
struct ZNIrrep{N} <: AbstractIrrep{ℤ{N}}
    n::UInt8
    function ZNIrrep{N}(n::Integer) where {N}
        N ≤ SMALL_ZN_CUTOFF || throw(DomainError(N, "N exceeds the maximal value,
        use `LargeZNIrrep` instead"))
        return new{N}(UInt8(mod(n, N)))
    end
end
struct LargeZNIrrep{N} <: AbstractIrrep{ℤ{N}}
    n::UInt
    function LargeZNIrrep{N}(n::Integer) where {N}
        N ≤ (typemax(UInt) ÷ 2) || throw(DomainError(N, "N exceeds the maximal
        value"))
        return new{N}(UInt(mod(n, N)))
    end

end
Base.getindex(::IrrepTable, ::Type{ℤ{N}}) where {N} = N ≤ SMALL_ZN_CUTOFF ?
ZNIrrep{N} : LargeZNIrrep{N}
...
```

and continues along simular lines of the `U1Irrep` implementation above, by replacing the arithmetic with modulo `N` arithmetic.

The storage benefits for small `N` are not only due to a smaller integer type in the sector itself, but emerges as a result of the following distinction in the iterator size:

```julia
Base.IteratorSize(::Type{SectorValues{<:ZNIrrep}}) = HasLength()
Base.IteratorSize(::Type{SectorValues{<:LargeZNIrrep}}) = SizeUnknown()
```

As a result, the `GradedSpace` implementation (see next section on Graded spaces) to store general direct sum objects $V = \bigoplus_a \mathbb{C}^{n_a} \otimes R_a$ will use a very different internal representation for those two cases.

We furthermore define some aliases for the first (and most commonly used `ℤ{N}` irreps)

```julia
const Z2Irrep = ZNIrrep{2}
const Z3Irrep = ZNIrrep{3}
const Z4Irrep = ZNIrrep{4}
```

which we can illustrate via

```julia
julia> z = Z3Irrep(1)
Irrep[ℤ₃](1)

julia> ZNIrrep{3}(1) ⊗ Irrep[ℤ₃](1)
(Irrep[ℤ₃](2),)

julia> dual(z)
Irrep[ℤ₃](2)

julia> unit(z)
Irrep[ℤ₃](0)
```

As a final remark on the irreps of abelian groups, note that even though `a ⊗ b` is equivalent to a single new label `c`, we return this result as an iterable container, in this case a one-element tuple `(c,)`.

The first example of irreps of a non-abelian group is that of SU_2, the implementation of which is summarized by

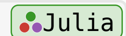
```julia
struct SU2Irrep <: AbstractIrrep{SU₂}
    j::HalfInt
    function SU2Irrep(j)
        j >= zero(j) || error("Not a valid SU₂ irrep")
        return new(j)
    end
end
Base.getindex(::IrrepTable, ::Type{SU₂}) = SU2Irrep
Base.convert(::Type{SU2Irrep}, j::Real) = SU2Irrep(j)

# basic properties
const _su2one = SU2Irrep(zero(HalfInt))
unit(::Type{SU2Irrep}) = _su2one
dual(s::SU2Irrep) = s
dim(s::SU2Irrep) = twice(s.j) + 1
```

```
Base.isless(s1::SU2Irrep, s2::SU2Irrep) = isless(s1.j, s2.j)

# fusion product iterator
const SU2IrrepProdIterator = SectorProductIterator{SU2Irrep}
Base.IteratorSize(::Type{SU2IrrepProdIterator}) = Base.HasLength()
Base.length(it::SU2IrrepProdIterator) = length(abs(it.a.j - it.b.j):(it.a.j +
it.b.j))
function Base.iterate(it::SU2IrrepProdIterator, state = abs(it.a.j - it.b.j))
    return state > (it.a.j + it.b.j) ? nothing : (SU2Irrep(state), state + 1)
end

# fusion and braidingdata
FusionStyle(::Type{SU2Irrep}) = SimpleFusion()
sectorscalartype(::Type{SU2Irrep}) = Float64

Nsymbol(sa::SU2Irrep, sb::SU2Irrep, sc::SU2Irrep) = WignerSymbols.δ(sa.j, sb.j,
sc.j)
function Fsymbol(
        s1::SU2Irrep, s2::SU2Irrep, s3::SU2Irrep,
        s4::SU2Irrep, s5::SU2Irrep, s6::SU2Irrep
    )
    if all(==(_su2one), (s1, s2, s3, s4, s5, s6))
        return 1.0
    else
        return sqrtdim(s5) * sqrtdim(s6) *
            WignerSymbols.racahW(
            sectorscalartype(SU2Irrep), s1.j, s2.j, s4.j, s3.j,
            s5.j, s6.j
        )
    end
end
function Rsymbol(sa::SU2Irrep, sb::SU2Irrep, sc::SU2Irrep)
    Nsymbol(sa, sb, sc) || return zero(sectorscalartype(SU2Irrep))
    return iseven(convert(Int, sa.j + sb.j - sc.j)) ?
    one(sectorscalartype(SU2Irrep)) :
        -one(sectorscalartype(SU2Irrep))
end

# values iterator
Base.IteratorSize(::Type{SectorValues{SU2Irrep}}) = IsInfinite()
Base.iterate(::SectorValues{SU2Irrep}, i::Int = 0) = (SU2Irrep(half(i)), i + 1)
function Base.getindex(::SectorValues{SU2Irrep}, i::Int)
    return 1 <= i ? SU2Irrep(half(i - 1)) : throw(BoundsError(values(SU2Irrep),
    i))
end
```

```
findindex(::SectorValues{SU2Irrep}, s::SU2Irrep) = twice(s.j) + 1

# hashing
Base.hash(s::SU2Irrep, h::UInt) = hash(s.j, h)
```

and some methods for pretty printing and converting from real numbers to irrep labels. Here, the fusion rules are implemented lazily using the `SectorProductIterator` defined above. Furthermore, the topological data (i.e. `Nsymbol` and `Fsymbol`) are provided by the package WignerSymbols.jl. Note that, while WignerSymbols.jl is able to generate the required data in arbitrary precision, we have explicitly restricted the scalar type of `SU2Irrep` to `Float64` for efficiency.

The following example illustrates the usage of `SU2Irrep`



```
julia> s = SU2Irrep(3//2)
Irrep[SU₂](3/2)

julia> dual(s)
Irrep[SU₂](3/2)

julia> dim(s)
4

julia> collect(s ⊗ s)
4-element Vector{SU2Irrep}:
 0
 1
 2
 3

julia> for s2 in s ⊗ s
           @show s2
           @show Nsymbol(s, s, s2)
           @show Rsymbol(s, s, s2)
       end
s2 = Irrep[SU₂](0)
Nsymbol(s, s, s2) = true
Rsymbol(s, s, s2) = -1.0
s2 = Irrep[SU₂](1)
Nsymbol(s, s, s2) = true
Rsymbol(s, s, s2) = 1.0
s2 = Irrep[SU₂](2)
Nsymbol(s, s, s2) = true
Rsymbol(s, s, s2) = -1.0
s2 = Irrep[SU₂](3)
Nsymbol(s, s, s2) = true
Rsymbol(s, s, s2) = 1.0
```

Other non-abelian groups for which the irreps are implemented are the dihedral groups D_N, the alternating group of order four A_4 and the semidirect product $U_1 \ltimes \mathbb{Z}_2$. In the context of quantum systems, the latter occurs in the case of systems with particle hole symmetry and the non-trivial element of $\mathbb{Z}_2$ acts as charge conjugation C. It has the effect of interchanging U_1 irreps n and $-n$, and turns them together in a joint two-dimensional index, except for the case $n = 0$. Irreps are therefore labeled by integers $n \geqq 0$, however for $n = 0$ the $\mathbb{Z}_2$ symmetry can be realized trivially or non-trivially, resulting in an even and odd one-dimensional irrep with U_1 charge 0. Given $U_1 \approx SO_2$, this group is also simply known as O_2, and the two representations with $n = 0$ are the scalar and pseudo-scalar, respectively. However, because we also allow for half integer representations, we refer to it as `Irrep[CU₁]` or `CU1Irrep` in full.

```julia
struct CU1Irrep <: AbstractIrrep{CU₁}
    j::HalfInt # value of the U1 charge
    s::Int # rep of charge conjugation:
    # if j == 0, s = 0 (trivial) or s = 1 (non-trivial),
    # else s = 2 (two-dimensional representation)
    # Let constructor take the actual half integer value j
    function CU1Irrep(j::Real, s::Int = ifelse(j>zero(j), 2, 0))
        if ((j > zero(j) && s == 2) || (j == zero(j) && (s == 0 || s == 1)))
            new(j, s)
        else
            error("Not a valid CU₁ irrep")
        end
    end
end

unit(::Type{CU1Irrep}) = CU1Irrep(zero(HalfInt), 0)
dual(c::CU1Irrep) = c
dim(c::CU1Irrep) = ifelse(c.j == zero(HalfInt), 1, 2)

FusionStyle(::Type{CU1Irrep}) = SimpleFusion()
...
```

The rest of the implementation can be read in the source code, but is rather long due to all the different cases for the arguments of `Fsymbol` . For the dihedrial groups D_N, which can be intepreted as the semidirect product $\mathbb{Z}_N \ltimes \mathbb{Z}_2$, the representation theory is obtained quite similarly, and is implmented as the type `DNIrrep{N}` .

Of the aforementioned groups, only A_4 has a representation theory for which `FusionStyle(I) == GenericFusion()` , i.e. where fusion mulitplicities are required. Another example where this does appear is for the irreps of `SU{N}` for $N > 2$. Such sectors are supported through SUNRepresentations.jl, which implements numerical routines to compute the topological data of the representation theory of these groups, as no general analytic formula is available.

6.5 Combining different sectors

It is also possible to combine two or more different types of symmetry sectors, e.g. when the total symmetry group is a direct product of individual simple groups. Such combined sectors are obtained

using the binary operator ⊠ , which can be entered as `\boxtimes` +TAB. The resulting type is called `ProductSector` , which simply wraps the individual sectors, but knows how to combine their fusion and braiding data correctly. First some examples

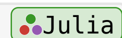
```julia
julia> a = Z3Irrep(1) ⊠ Irrep[U₁](1)
Irrep[ℤ₃ × U₁](1, 1)

julia> typeof(a)
ProductSector{Tuple{Z3Irrep, U1Irrep}}

julia> dual(a)
Irrep[ℤ₃ × U₁](2, -1)

julia> unit(a)
Irrep[ℤ₃ × U₁](0, 0)

julia> dim(a)
1

julia> collect(a ⊗ a)
1-element Vector{ProductSector{Tuple{Z3Irrep, U1Irrep}}}:
 (2, 2)

julia> FusionStyle(a)
UniqueFusion()

julia> b = Irrep[ℤ₃](1) ⊠ Irrep[SU₂](3//2)
Irrep[ℤ₃ × SU₂](1, 3/2)

julia> typeof(b)
ProductSector{Tuple{Z3Irrep, SU2Irrep}}

julia> dual(b)
Irrep[ℤ₃ × SU₂](2, 3/2)

julia> unit(b)
Irrep[ℤ₃ × SU₂](0, 0)

julia> dim(b)
4

julia> collect(b ⊗ b)
1×4 Matrix{ProductSector{Tuple{Z3Irrep, SU2Irrep}}}:
 (2, 0)  (2, 1)  (2, 2)  (2, 3)

julia> FusionStyle(b)
```

```
SimpleFusion()

julia> c = Irrep[SU₂](1) ⊠ SU2Irrep(3//2)
Irrep[SU₂ × SU₂](1, 3/2)

julia> typeof(c)
ProductSector{Tuple{SU2Irrep, SU2Irrep}}

julia> dual(c)
Irrep[SU₂ × SU₂](1, 3/2)

julia> unit(c)
Irrep[SU₂ × SU₂](0, 0)

julia> dim(c)
12

julia> collect(c ⊗ c)
3×4 Matrix{ProductSector{Tuple{SU2Irrep, SU2Irrep}}}:
 (0, 0)  (0, 1)  (0, 2)  (0, 3)
 (1, 0)  (1, 1)  (1, 2)  (1, 3)
 (2, 0)  (2, 1)  (2, 2)  (2, 3)

julia> FusionStyle(c)
SimpleFusion()
```

We refer to the source file of `ProductSector` for implementation details.

The symbol ⊠ refers to the Deligne tensor product within the literature on category theory. Indeed, the category of representation of a product group `G₁ × G₂` corresponds to the Deligne tensor product of the categories of representations of the two groups separately. But this definition also extends to other categories which are not associated with the representation theory of a group, as discussed below. Note that ⊠ also works in the type domain, i.e. `Irrep[ℤ₃] ⊠ Irrep[CU₁]` can be used to create `ProductSector{Tuple{Irrep[ℤ₃], Irrep[CU₁]}}`. Instances of this type can be constructed by giving a number of arguments, where the first argument is used to construct the first sector, and so forth. Furthermore, for representations of groups, we also enabled the notation `Irrep[ℤ₃ × CU₁]`, with × obtained using `\times+TAB`. However, this is merely for convenience, as `Irrep[ℤ₃] ⊠ Irrep[CU₁]` is not a subtype of the abstract type `AbstractIrrep{ℤ₃ × CU₁}`. As is often the case with the Julia type system, the purpose of subtyping `AbstractIrrep` was to share common functionality and thereby simplify the implementation of irreps of the different groups discussed above, but not to express a mathematical hierarchy.

Some more examples:



```
julia> a = Z3Irrep(1) ⊠ Irrep[CU₁](1.5)
Irrep[ℤ₃ × CU₁](1, (3/2, 2))

julia> a isa Irrep[ℤ₃] ⊠ CU1Irrep
```

```
true

julia> a isa Irrep[ℤ₃ × CU₁]
true

julia> a isa AbstractIrrep{ℤ₃ × CU₁}
false

julia> a == Irrep[ℤ₃ × CU₁](1, 1.5)
true
```

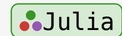
6.6 Defining a new type of sector

By now, it should be clear how to implement a new `Sector` subtype. Ideally, a new `I <: Sector` type is a `struct I ... end` (immutable) that has `isbitstype(I) == true` (see Julia's manual), and implements the following minimal set of methods

```julia
TensorKit.unit(::Type{I}) = I(...)
TensorKit.dual(a::I) = I(...)
Base.isless(a::I, b::I)

TensorKit.FusionStyle(::Type{I}) = ... # UniqueFusion(), SimpleFusion(),
GenericFusion()
TensorKit.Nsymbol(a::I, b::I, c::I) = ... # Bool or Integer if FusionStyle(I) ==
GenericFusion()

TensorKit.:⊗(a::I, b::I) = ... # some iterable object that generates all possible
fusion outputs
# or
Base.iterate(::SectorProductIterator{I}[, state]) = ...
Base.IteratorSize(::Type{SectorProductIterator{I}}) = ... # HasLength() or
IsInfinite()
Base.length(::SectorProductIterator{I}) = ... # if previous function returns
HasLength()

TensorKit.Fsymbol(a::I, b::I, c::I, d::I, e::I, f::I) = ...

TensorKit.BraidingStyle(::Type{I}) = ... # NoBraiding(), Bosonic(), Fermionic(),
Anyonic()
TensorKit.Rsymbol(a::I, b::I, c::I) = ... # only if BraidingStyle(I) !=
NoBraiding()

Base.iterate(::TensorKit.SectorValues{I}[, state]) = ...
Base.IteratorSize(::Type{TensorKit.SectorValues{I}}) = ... # HasLenght() or
IsInfinite()
# if previous function returns HasLength():
Base.length(::TensorKit.SectorValues{I}) = ...
```

```
# optional, but recommended if IteratorSize returns HasLength():
Base.getindex(::TensorKit.SectorValues{I}, i::Int) = ...
TensorKit.findindex(::TensorKit.SectorValues{I}, c::I) = ...

Base.hash(a::I, h::UInt)
```

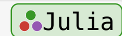
Additionally, suitable definitions can be given for



```
TensorKit.sectorscalartype(::Type{I}) = ... # Int, Float64,
ComplexF64, ...
TensorKit.dim(a::I) = ...
TensorKit.frobeniusschur_phase(a::I) = ...
TensorKit.Bsymbol(a::I, b::I, c::I) = ...
```

6.7 Fermionic sectors

All of the sectors discussed in Group representations have a bosonic braiding style. This does not mean that `Rsymbol` is always trivial, as for example for `SU2Irrep` the definition was given by

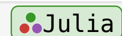


```
function Rsymbol(sa::SU2Irrep, sb::SU2Irrep, sc::SU2Irrep)
    Nsymbol(sa, sb, sc) || return zero(sectorscalartype(SU2Irrep))
    return iseven(convert(Int, sa.j + sb.j - sc.j)) ?
    one(sectorscalartype(SU2Irrep)) :
        -one(sectorscalartype(SU2Irrep))
end
```

It does however mean that all twists θ_a are trivial (equal to 1). We refer to the appendix on Category theory for more details on the meaning of the twist. In summary, triviality of the twists implies that self-crossings of lines in tensor diagrams can be ignored, i.e. they can be removed without changing the value of the diagram.

As is well known, this becomes more subtle when fermionic degrees are involved. Technically, fermions are described using super vector spaces, which are $\mathbb{Z}_2$-graded vector spaces $V = V_0 \oplus V_1$, i.e. the vector space is decomposed as an (orthogonal) direct sum into an even and odd subspace, corresponding to states with even and odd fermion parity, respectively. The tensor product of two super vector spaces V and W is again graded as $(V \otimes W)_0 = (V_0 \otimes W_0) \oplus (V_1 \otimes W_1)$ and $(V \otimes W)_1 = (V_0 \otimes W_1) \oplus (V_1 \otimes W_0)$. However, when exchanging two super vector spaces in such a tensor product, the natural isomorphism $V \otimes W \to W \otimes V$ takes into account the fermionic nature by acting with a minus sign in the subspace $V_1 \otimes W_1$. This is known as the Koszul sign rule.

The super vector space structure fits naturally in the framework of TensorKit.jl. Indeed, the grading naturally corresponds to a $\mathbb{Z}_2$-valued sector structure, which we implement as `FermionParity`:



```
struct FermionParity <: Sector
    isodd::Bool
end
const fℤ₂ = FermionParity
fermionparity(f::FermionParity) = f.isodd
```

with straightforward fusion rules and associators

```julia
⊗(a::FermionParity, b::FermionParity) = (FermionParity(a.isodd ⊻
b.isodd),)
function Nsymbol(a::FermionParity, b::FermionParity, c::FermionParity)
    return (a.isodd ⊻ b.isodd) == c.isodd
end
function Fsymbol(a::I, b::I, c::I, d::I, e::I, f::I) where {I <: FermionParity}
    return Int(Nsymbol(a, b, e) * Nsymbol(e, c, d) * Nsymbol(b, c, f) * Nsymbol(a,
    f, d))
end
```

but with non-trivial braiding and twist

```julia
function Rsymbol(a::I, b::I, c::I) where {I <: FermionParity}
    return a.isodd && b.isodd ? -Int(Nsymbol(a, b, c)) : Int(Nsymbol(a, b, c))
end
twist(a::FermionParity) = a.isodd ? -1 : +1
```

The super vector space structure can also be combined with other sector types using the ⊠ operator discussed above. In some cases, there is a richer symmetry than $\mathbb{Z}_2$ associated with the fermionic degrees of freedom, and there is a natural fermion parity associated with the sectors of that symmetry. An example would be a U_1 symmetry associated with fermion number conservation, where odd U_1 charges correspond to odd fermion parity. However, it is then always possible to separate out the fermion parity structure as a separate sector, and treat the original sectors as bosonic, by only restricting to combinations of sectors that satisfy the natural fermion parity association.

For convenience (and partially due to legacy reasons), TensorKitSectors.jl does provide `FermionNumber` and `FermionSpin` constructors, which are defined as

```julia
const FermionNumber = U1Irrep ⊠ FermionParity
const fU₁ = FermionNumber
FermionNumber(a::Int) = U1Irrep(a) ⊠ FermionParity(isodd(a))

const FermionSpin = SU2Irrep ⊠ FermionParity
const fSU₂ = FermionSpin
FermionSpin(j::Real) = (s = SU2Irrep(j); s ⊠ FermionParity(isodd(twice(s.j))))
```

We conclude this subsection with some examples.

```julia
julia> p = FermionParity(true)
FermionParity(1)

julia> p ⊗ p
(FermionParity(0),)

julia> twist(p)
-1

julia> FusionStyle(p)
UniqueFusion()
```

```
julia> BraidingStyle(p)
Fermionic()

julia> s = FermionSpin(3//2)
(Irrep[SU₂](3/2) ⊠ FermionParity(1))

julia> dim(s)
4

julia> twist(s)
-1.0

julia> typeof(s)
ProductSector{Tuple{SU2Irrep, FermionParity}}

julia> FusionStyle(s)
SimpleFusion()

julia> BraidingStyle(s)
Fermionic()

julia> collect(s ⊗ s)
4×1 Matrix{ProductSector{Tuple{SU2Irrep, FermionParity}}}:
 (0, 0)
 (1, 0)
 (2, 0)
 (3, 0)

julia> for s2 in s ⊗ s
           @show s2
           @show Rsymbol(s, s, s2)
       end
s2 = (Irrep[SU₂](0) ⊠ FermionParity(0))
Rsymbol(s, s, s2) = 1.0
s2 = (Irrep[SU₂](1) ⊠ FermionParity(0))
Rsymbol(s, s, s2) = -1.0
s2 = (Irrep[SU₂](2) ⊠ FermionParity(0))
Rsymbol(s, s, s2) = 1.0
s2 = (Irrep[SU₂](3) ⊠ FermionParity(0))
Rsymbol(s, s, s2) = -1.0
```

Note in particular how the `Rsymbol` values have opposite signs to the bosonic case, where the fusion of two equal half-integer spins to the trivial sector is antisymmetric and would thus have `Rsymbol` value `-1`.

6.8 Anyons

Both `Bosonic` and `Fermionic` braiding styles are `SymmetricBraiding` styles, which means that exchanging two sectors twice is equivalent to the identity operation. In tensor network diagrams, this implies that lines that cross twice are equivalent to them not crossing at all, or also, that there is no distinction betweeen a line crossing “above” or “below” another line. More technically, the relevant group describing the exchange processes is the permutation group, whereas in more general cases it would be the braid group.

This more general case is denoted as the `Anyonic` braiding style in TensorKit.jl, because examples of this behaviour appear in the context of anyons in topological phases of matter.

There are currently two well-known sector types with `Anyonic` braiding style implemented in TensorKitSectors. l, namely `FibonacciAnyon` and `IsingAnyon`. Their values represent the (equivalence classes of) simple objects of the well-known Fibonicci and Ising fusion categories. As an example, we illustrate below the Fibonacci anyons, which has only two distinct sectors, namely the unit sector `𝟙` and one non-trivial sector denoted as `τ`. The fusion rules are given by `τ ⊗ τ = 𝟙 ⊕ τ`, and the topological data is summarized by the following code

```julia
julia> 𝟙 = FibonacciAnyon(:I)
FibonacciAnyon(:I)

julia> τ = FibonacciAnyon(:τ)
FibonacciAnyon(:τ)

julia> collect(τ ⊗ τ)
2-element Vector{FibonacciAnyon}:
 :I
 :τ

julia> FusionStyle(τ)
SimpleFusion()

julia> BraidingStyle(τ)
Anyonic()

julia> dim(𝟙)
1.0

julia> dim(τ)
1.618033988749895

julia> F𝟙 = Fsymbol(τ,τ,τ,𝟙,τ,τ)
1.0

julia> Fτ = [Fsymbol(τ,τ,τ,τ,𝟙,𝟙) Fsymbol(τ,τ,τ,τ,𝟙,τ); Fsymbol(τ,τ,τ,τ,τ,𝟙)
Fsymbol(τ,τ,τ,τ,τ,τ)]
2×2 Matrix{Float64}:
```

```
 0.618034   0.786151
 0.786151  -0.618034

julia> Fτ'*Fτ
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0

julia> polar(x) = rationalize.((abs(x), angle(x)/(2pi)))
polar (generic function with 1 method)

julia> Rsymbol(τ,τ,𝟙) |> polar
(1//1, 2//5)

julia> Rsymbol(τ,τ,τ) |> polar
(1//1, -3//10)

julia> twist(τ) |> polar
(1//1, -2//5)
```

6.9 Further generalizations

The `Anyonic` braiding style is one generalization beyond the bosonic and fermionic representation theory of groups, i.e. the action of groups on vector spaces and super vector spaces. It is also possible to consider fusion categories without braiding structure, represented as `NoBraiding` in TensorKitSectors.jl. Indeed, the framework for sectors outlined above is in one-to-one correspondence to the topological data for specifying a unitary (spherical and braided, and hence ribbon) fusion category, which is reviewed in the appendix on category theory. For such categories, the objects are not necessarily vector spaces and the fusion and splitting tensors $X^{ab}_{c,\mu}$ do not necessarily exist as actual tensors. However, the morphism spaces $c \to a \otimes b$ still behave as vector spaces, and the $X^{ab}_{c,\mu}$ act as generic basis for that space. As TensorKit.jl does not rely on the $X^{ab}_{c,\mu}$ themselves (even when they do exist), it can also deal with such general fusion categories. An extensive list of (the topological data of) such fusion categories, with and without braiding, is provided in CategoryData.jl.

Within TensorKit.jl, the only sector with `NoBraiding` is the `PlanarTrivial` sector, which is actually equivalent to the `Trivial` sector, but where the braiding has been “disabled” for testing purposes.

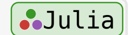
Finally, as mentioned above, a recent extension prepares TensorKitSectors.jl to deal with multi-fusion categories, where the sectors (simple objects) are organized in a matrix-like structure and thus have an additional row and column index. Fusion between sectors is only possible when the row and column indices match appropriately; otherwise the fusion product is empty. In this structure, the different *diagonal* sectors define separate fusion categories, whereas the *off-diagonal* sectors define bimodule categories between these fusion categories. Every diagonal set of sectors has its own unit sector, which also acts as the left / right unit for other sectors in the same column / row. The global unit object is not simple, but rather given by the direct sum of all diagonal unit sectors. We do not document or illustrate this structure here, but refer to the relevant functions `leftunit` , `rightunit` , `allunits` and `UnitStyle` for more information. Furthermore, we refer to MultiTensorKit.jl for examples and ongoing development work on using multi-fusion categories.

Chapter 7

Graded spaces

We have introduced `Sector` subtypes as a way to label the irreps or sectors in the decomposition $V = \bigoplus_a \mathbb{C}^{n_a} \otimes R_a$. To actually represent such spaces, we now also introduce a corresponding type `GradedSpace` , which is a subtype of `ElementarySpace` :

`TensorKit.GradedSpace` – Type.



```julia
struct GradedSpace{I<:Sector, D} <: ElementarySpace
GradedSpace{I,D}(dims; dual::Bool = false) where {I<:Sector, D}
```

A complex Euclidean space with a grading, i.e. a direct sum structure corresponding to labels in a set `I` , the objects of which have the structure of a monoid with respect to a monoidal product `⊗` . In practice, we restrict the label set to be a set of superselection sectors of type `I<:Sector` , e.g. the set of distinct irreps of a finite or compact group, or the isomorphism classes of simple objects of a unitary and pivotal (pre-, multi-) fusion category.

Here `dims` represents the degeneracy or multiplicity of every sector.

The data structure `D` of `dims` will depend on the result `Base.IteratorSize(values(I))` . If the result is of type `HasLength` or `HasShape` , `dims` will be stored in a `NTuple{N,Int}` with `N = length(values(I))` . This requires that a sector `s::I` can be transformed into an index via `s == getindex(values(I), i)` and `i == findindex(values(I), s)` . If `Base.IteratorElsize(values(I))` results `IsInfinite()` or `SizeUnknown()` , a `SectorDict{I,Int}` is used to store the non-zero degeneracy dimensions with the corresponding sector as key. The parameter `D` is hidden from the user and should typically be of no concern.

The concrete type `GradedSpace{I,D}` with correct `D` can be obtained as `Vect[I]` , or if `I == Irrep[G]` for some `G<:Group` , as `Rep[G]` .

source

Here, `D` is a type parameter to denote the data structure used to store the degeneracy or multiplicity dimensions n_a of the different sectors. For convenience, `Vect[I]` will return the fully concrete type with `D` specified.

Note that, conventionally, a graded vector space is a space that has a natural direct sum decomposition over some set of labels, i.e. $V = \bigoplus_{a \in I} V_a$ where the label set I has the structure of a semigroup $a \otimes b = c \in I$. Here, we generalize this notation by using for I the fusion ring of a fusion category, $a \otimes b = \bigoplus_{c \in I} \bigoplus_{\mu=1}^{N_{a,b}^c} c$. However, this is mostly to lower the barrier, as really the instances of `GradedSpace` represent just general objects in a fusion category (or strictly speaking, a pre-fusion category, as we allow for an infinite number of simple objects, e.g. the irreps of a continuous group).

7.1 Implementation details

As mentioned, the way in which the degeneracy dimensions n_a are stored depends on the specific sector type `I` , more specifically on the `IteratorSize` of `values(I)` . If `IteratorSize(values(I))` `isa Union{IsInfinite, SizeUnknown}` , the different sectors a and their corresponding degeneracy

n_a are stored as key value pairs in an `Associative` array, i.e. a dictionary `dims::SectorDict`. As the total number of sectors in `values(I)` can be infinite, only sectors a for which n_a are stored. Here, `SectorDict` is a constant type alias for a specific dictionary implementation, which currently resorts to `SortedVectorDict` implemented in TensorKit.jl. Hence, the sectors and their corresponding dimensions are stored as two matching lists (`Vector` instances), which are ordered based on the property `isless(a::I, b::I)`. This ensures that the space $V = \bigoplus_a \mathbb{C}^{n_a} \otimes R_a$ has some unique canonical order in the direct sum decomposition, such that two different but equal instances created independently always match.

If `IteratorSize(values(I)) isa Union{HasLength, HasShape}`, the degeneracy dimensions `n_a` are stored for all sectors `a ∈ values(I)` (also if `n_a == 0`) in a tuple, more specifically a `NTuple{N, Int}` with `N = length(values(I))`. The methods `getindex(values(I), i)` and `findindex(values(I), a)` are used to map between a sector `a ∈ values(I)` and a corresponding index `i ∈ 1:N`. As `N` is a compile time constant, these types can be created in a type stable manner. Note however that this implies that for large values of `N`, it can be beneficial to define `IteratorSize(values(a)) = SizeUnknown()` to not overly burden the compiler.

7.2 Constructing instances

As mentioned, the convenience method `Vect[I]` will return the concrete type `GradedSpace{I, D}` with the matching value of `D`, so that should never be a user's concern. In fact, for consistency, `Vect[Trivial]` will just return `ComplexSpace`, which is not even a specific type of `GradedSpace`. For the specific case of group irreps as sectors, one can use `Rep[G]` with `G` the group, as inspired by the categorical name $\mathbf{Rep}_G$. Some illustrations:

```julia
julia> Vect[Trivial]
ComplexSpace

julia> Vect[U1Irrep]
GradedSpace{U1Irrep, TensorKit.SortedVectorDict{U1Irrep, Int64}}

julia> Vect[Irrep[U₁]]
GradedSpace{U1Irrep, TensorKit.SortedVectorDict{U1Irrep, Int64}}

julia> Rep[U₁]
GradedSpace{U1Irrep, TensorKit.SortedVectorDict{U1Irrep, Int64}}

julia> Rep[ℤ₂ × SU₂]
GradedSpace{ProductSector{Tuple{Z2Irrep, SU2Irrep}},
TensorKit.SortedVectorDict{ProductSector{Tuple{Z2Irrep, SU2Irrep}}, Int64}}

julia> Vect[Irrep[ℤ₂ × SU₂]]
GradedSpace{ProductSector{Tuple{Z2Irrep, SU2Irrep}},
TensorKit.SortedVectorDict{ProductSector{Tuple{Z2Irrep, SU2Irrep}}, Int64}}
```

Note that we also have the specific alias `U₁Space`. In fact, for all the common groups we have a number of aliases, both in ASCII and using Unicode:

```julia
# ASCII type aliases
```

```
const ZNSpace{N} = GradedSpace{ZNIrrep{N}, NTuple{N,Int}}
const Z2Space = ZNSpace{2}
const Z3Space = ZNSpace{3}
const Z4Space = ZNSpace{4}
const U1Space = Rep[U₁]
const CU1Space = Rep[CU₁]
const SU2Space = Rep[SU₂]

# Unicode alternatives
const ℤ₂Space = Z2Space
const ℤ₃Space = Z3Space
const ℤ₄Space = Z4Space
const U₁Space = U1Space
const CU₁Space = CU1Space
const SU₂Space = SU2Space
```

To create specific instances of those types, one can e.g. just use `V = GradedSpace(a => n_a, b => n_b, c => n_c)` or `V = GradedSpace(iterator)` where `iterator` is any iterator (e.g. a dictionary or a generator) that yields `Pair{I, Int}` instances. With those constructions, `I` is inferred from the type of sectors. However, it is often more convenient to specify the sector type explicitly (using one of the many alias provided), since then the sectors are automatically converted to the correct type. Thereto, one can use `Vect[I]` , or when `I` corresponds to the irreducible representations of a group, `Rep[G]` . Some examples:

```julia
julia> Vect[Irrep[U₁]](0 => 3, 1 => 2, -1 => 1) ==
         GradedSpace(U1Irrep(0) => 3, U1Irrep(1) => 2, U1Irrep(-1) => 1) ==
         U1Space(0 => 3, 1 => 2, -1 => 1)
true
```

The fact that `Rep[G]` also works with product groups makes it easy to specify e.g.

```julia
julia> Rep[ℤ₂ × SU₂]((0, 0) => 3, (1, 1/2) => 2, (0, 1) => 1) ==
         GradedSpace((Z2Irrep(0) ⊠ SU2Irrep(0)) => 3, (Z2Irrep(1) ⊠
         SU2Irrep(1/2)) => 2, (Z2Irrep(0) ⊠ SU2Irrep(1)) => 1)
true
```

7.3 Methods

There are a number of methods to work with instances `V` of `GradedSpace` . The function `sectortype` returns the type of the sector labels. It also works on other vector spaces, in which case it returns `Trivial` . The function `sectors` returns an iterator over the different sectors `a` with non-zero `n_a` , for other `ElementarySpace` types it returns `(Trivial,)` . The degeneracy dimensions `n_a` can be extracted as `dim(V, a)` , it properly returns `0` if sector `a` is not present in the decomposition of `V` . With `hassector(V, a)` one can check if `V` contains a sector `a` with `dim(V, a) > 0` . Finally, `dim(V)` returns the total dimension of the space `V` , i.e. $\sum_a n_a d_a$ or thus `dim(V) = sum(dim(V, a) * dim(a) for a in sectors(V))` . Note that a representation space `V` has certain sectors `a` with dimensions `n_a` , then its dual `V'` will report to have sectors `dual(a)` , and `dim(V', dual(a)) == n_a` . There is a subtlety regarding the difference between the dual of a representation space R_a^*, on which the

conjugate representation acts, and the representation space of the irrep `dual(a) == conj(a)` that is isomorphic to the conjugate representation, i.e. $R_{\overline{a}} \approx R_a^*$ but they are not equal. We return to this in the section on fusion trees. This is true also in more general fusion categories beyond the representation categories of groups.

Other methods for `ElementarySpace`, such as `dual`, `fuse` and `flip` also work. In fact, `GradedSpace` is the reason `flip` exists, because in this case it is different than `dual`. The existence of flip originates from the non-trivial isomorphism between $R_{\overline{a}}$ and R_a^*, i.e. the representation space of the dual $\overline{a}$ of sector a and the dual of the representation space of sector a. In order for `flip(V)` to be isomorphic to `V`, it is such that, if `V = GradedSpace(a=>n_a,...)` then `flip(V) = dual(GradedSpace(dual(a)=>n_a,....))`.

Furthermore, for two spaces `V1 = GradedSpace(a => n1_a, ...)` and `V2 = GradedSpace(a => n2_a, ...)`, we have `infimum(V1, V2) = GradedSpace(a => min(n1_a, n2_a), ....)` and similarly for `supremum`, i.e. they act on the degeneracy dimensions of every sector separately. Therefore, it can be that the return value of `infimum(V1, V2)` or `supremum(V1, V2)` is neither equal to `V1` or `V2`.

For `W` a `ProductSpace{Vect[I], N}`, `sectors(W)` returns an iterator that generates all possible combinations of sectors `as` represented as `NTuple{I, N}`. The function `dims(W, as)` returns the corresponding tuple with degeneracy dimensions, while `dim(W, as)` returns the product of these dimensions. `hassector(W, as)` is equivalent to `dim(W, as) > 0`. Finally, there is the function `blocksectors(W)` which returns a list (of type `Vector`) with all possible “block sectors” or total/coupled sectors that can result from fusing the individual uncoupled sectors in `W`. Correspondingly, `blockdim(W, a)` counts the total degeneracy dimension of the coupled sector `a` in `W`. The machinery for computing this is the topic of the next section on Fusion trees, but first, it’s time for some examples.

7.4 Examples

Let’s start with an example involving U_1:

```
julia> V1 = Rep[U₁](0=>3, 1=>2, -1=>1)
Rep[U₁](…) of dim 6:
  0 => 3
  1 => 2
 -1 => 1

julia> V1 == U1Space(0=>3, 1=>2, -1=>1) == U₁Space(-1=>1, 1=>2,0=>3) # order
doesn't matter
true

julia> (sectors(V1)...,)
(Irrep[U₁](0), Irrep[U₁](1), Irrep[U₁](-1))

julia> dim(V1, U1Irrep(1))
2

julia> dim(V1', Irrep[U₁](1)) == dim(V1, conj(U1Irrep(1))) == dim(V1, U1Irrep(-1))
true

julia> hassector(V1, Irrep[U₁](1))
```

```
true

julia> hassector(V1, Irrep[U₁](2))
false

julia> dual(V1)
Rep[U₁](…)' of dim 6:
  0 => 3
  1 => 2
 -1 => 1

julia> flip(V1)
Rep[U₁](…)' of dim 6:
  0 => 3
  1 => 1
 -1 => 2

julia> dual(V1) ≅ V1
false

julia> flip(V1) ≅ V1
true

julia> V2 = U1Space(0=>2, 1=>1, -1=>1, 2=>1, -2=>1)
Rep[U₁](…) of dim 6:
  0 => 2
  1 => 1
 -1 => 1
  2 => 1
 -2 => 1

julia> infimum(V1, V2)
Rep[U₁](…) of dim 4:
  0 => 2
  1 => 1
 -1 => 1

julia> supremum(V1, V2)
Rep[U₁](…) of dim 8:
  0 => 3
  1 => 2
 -1 => 1
  2 => 1
 -2 => 1
```

```
julia> ⊕(V1,V2)
Rep[U₁](…) of dim 12:
  0 => 5
  1 => 3
 -1 => 2
  2 => 1
 -2 => 1

julia> W = ⊗(V1,V2)
(Rep[U₁](0 => 3, 1 => 2, -1 => 1) ⊗ Rep[U₁](0 => 2, 1 => 1, -1 => 1, 2 => 1, -2 =>
1))

julia> collect(sectors(W))
3×5 Matrix{Tuple{Any, Any}}:
 (Irrep[U₁](0), Irrep[U₁](0))   …  (Irrep[U₁](0), Irrep[U₁](-2))
 (Irrep[U₁](1), Irrep[U₁](0))      (Irrep[U₁](1), Irrep[U₁](-2))
 (Irrep[U₁](-1), Irrep[U₁](0))     (Irrep[U₁](-1), Irrep[U₁](-2))

julia> dims(W, (Irrep[U₁](0), Irrep[U₁](0)))
(3, 2)

julia> dim(W, (Irrep[U₁](0), Irrep[U₁](0)))
6

julia> hassector(W, (Irrep[U₁](0), Irrep[U₁](0)))
true

julia> hassector(W, (Irrep[U₁](2), Irrep[U₁](0)))
false

julia> fuse(W)
Rep[U₁](…) of dim 36:
  0 => 9
  1 => 8
 -1 => 7
  2 => 5
 -2 => 4
  3 => 2
 -3 => 1

julia> (blocksectors(W)...,)
(Irrep[U₁](0), Irrep[U₁](1), Irrep[U₁](-1), Irrep[U₁](2), Irrep[U₁](-2), Irrep[U₁]
(3), Irrep[U₁](-3))

julia> blockdim(W, Irrep[U₁](0))
```

```
9
```

and then with SU_2:

```julia
julia> V1 = Vect[Irrep[SU₂]](0=>3, 1//2=>2, 1=>1)
Rep[SU₂](…) of dim 10:
   0 => 3
 1/2 => 2
   1 => 1

julia> V1 == SU2Space(0=>3, 1/2=>2, 1=>1) == SU₂Space(0=>3, 0.5=>2, 1=>1)
true

julia> (sectors(V1)...,)
(Irrep[SU₂](0), Irrep[SU₂](1/2), Irrep[SU₂](1))

julia> dim(V1, SU2Irrep(1))
1

julia> dim(V1', SU2Irrep(1)) == dim(V1, conj(SU2Irrep(1))) == dim(V1, Irrep[SU₂](1))
true

julia> dim(V1)
10

julia> hassector(V1, Irrep[SU₂](1))
true

julia> hassector(V1, Irrep[SU₂](2))
false

julia> dual(V1)
Rep[SU₂](…)' of dim 10:
   0 => 3
 1/2 => 2
   1 => 1

julia> flip(V1)
Rep[SU₂](…)' of dim 10:
   0 => 3
 1/2 => 2
   1 => 1

julia> V2 = SU2Space(0=>2, 1//2=>1, 1=>1, 3//2=>1, 2=>1)
Rep[SU₂](…) of dim 16:
```

```
   0 => 2
 1/2 => 1
   1 => 1
 3/2 => 1
   2 => 1

julia> infimum(V1, V2)
Rep[SU₂](…) of dim 7:
   0 => 2
 1/2 => 1
   1 => 1

julia> supremum(V1, V2)
Rep[SU₂](…) of dim 19:
   0 => 3
 1/2 => 2
   1 => 1
 3/2 => 1
   2 => 1

julia> ⊕(V1,V2)
Rep[SU₂](…) of dim 26:
   0 => 5
 1/2 => 3
   1 => 2
 3/2 => 1
   2 => 1

julia> W = ⊗(V1,V2)
(Rep[SU₂](0 => 3, 1/2 => 2, 1 => 1) ⊗ Rep[SU₂](0 => 2, 1/2 => 1, 1 => 1, 3/2 => 1,
2 => 1))

julia> collect(sectors(W))
3×5 Matrix{Tuple{Any, Any}}:
 (Irrep[SU₂](0), Irrep[SU₂](0))    …  (Irrep[SU₂](0), Irrep[SU₂](2))
 (Irrep[SU₂](1/2), Irrep[SU₂](0))     (Irrep[SU₂](1/2), Irrep[SU₂](2))
 (Irrep[SU₂](1), Irrep[SU₂](0))       (Irrep[SU₂](1), Irrep[SU₂](2))

julia> dims(W, (Irrep[SU₂](0), Irrep[SU₂](0)))
(3, 2)

julia> dim(W, (Irrep[SU₂](0), Irrep[SU₂](0)))
6

julia> hassector(W, (SU2Irrep(0), SU2Irrep(0)))
```

```
true

julia> hassector(W, (SU2Irrep(2), SU2Irrep(0)))
false

julia> fuse(W)
Rep[SU₂](…) of dim 160:
   0 => 9
 1/2 => 11
   1 => 11
 3/2 => 9
   2 => 7
 5/2 => 3
   3 => 1

julia> (blocksectors(W)...,)
(Irrep[SU₂](0), Irrep[SU₂](1/2), Irrep[SU₂](1), Irrep[SU₂](3/2), Irrep[SU₂](2),
Irrep[SU₂](5/2), Irrep[SU₂](3))

julia> blockdim(W, SU2Irrep(0))
9
```

Chapter 8

Fusion trees

The gain in efficiency (both in memory occupation and computation time) obtained from using symmetric (equivariant) tensor maps is that, by Schur's lemma, they are block diagonal in the basis of coupled sectors, i.e. they exhibit block sparsity. To exploit this block diagonal form, it is however essential that we know the basis transformation from the individual (uncoupled) sectors appearing in the tensor product form of the domain and codomain, to the totally coupled sectors that label the different blocks. We refer to the latter as block sectors, as we already encountered in the previous section `blocksectors` and `blockdim` defined on the type `ProductSpace` .

This basis transformation consists of a basis of inclusion and projection maps, denoted as $X^{a_1 a_2 \ldots a_N}_{c,\alpha} : R_c \to R_{a_1} \otimes R_{a_2} \otimes \ldots \otimes R_{a_N}$ and their adjoints $(X^{a_1 a_2 \ldots a_N}_{c,\alpha})^\dagger$, such that

$$(X^{a_1 a_2 \ldots a_N}_{c,\alpha})^\dagger \circ X^{a_1 a_2 \ldots a_N}_{c',\alpha'} = \delta_{c,c'} \delta_{\alpha,\alpha'} \mathrm{id}_c$$

and

$$\sum_{c,\alpha} X^{a_1 a_2 \ldots a_N}_{c,\alpha} \circ (X^{a_1 a_2 \ldots a_N}_{c,\alpha})^\dagger = \mathrm{id}_{a_1 \otimes a_2 \otimes \ldots \otimes a_N} = \mathrm{id}_{a_1} \otimes \mathrm{id}_{a_2} \otimes \ldots \otimes \mathrm{id}_{a_N}$$

Fusion trees provide a particular way to construct such a basis. It is useful to know about the existence of fusion trees and how they are represented, as discussed in the first subsection. The next two subsections discuss possible manipulations that can be performed with fusion trees. These are used under the hood when manipulating the indices of tensors, but a typical user would not need to use these manipulations on fusion trees directly. Hence, these last two sections can safely be skipped.

8.1 Canonical representation

To couple or fuse the different sectors together into a single block sector, we can sequentially fuse together two sectors into a single coupled sector, which is then fused with the next uncoupled sector, using the splitting tensors $X^{c,\mu}_{a,b} : R_c \to R_a \otimes R_b$ and their adjoints. This amounts to the canonical choice of our tensor product, and for a given tensor mapping from $(((W_1 \otimes W_2) \otimes W_3) \otimes \ldots) \otimes W_{N_2})$ to $(((V_1 \otimes V_2) \otimes V_3) \otimes \ldots) \otimes V_{N_1})$, the corresponding fusion and splitting trees take the form

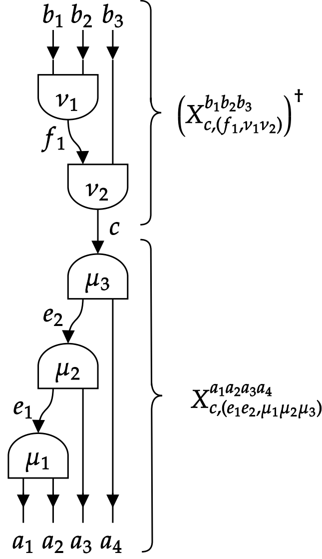


Figure 8.1: double fusion tree

for the specific case $N_1 = 4$ and $N_2 = 3$. We can separate this tree into the fusing part $(b_1 \otimes b_2) \otimes b_3 \to c$ and the splitting part $c \to (((a_1 \otimes a_2) \otimes a_3) \otimes a_4)$. Given that the fusion tree can be considered to be the adjoint of a corresponding splitting tree $c \to (b_1 \otimes b_2) \otimes b_3$, we now first consider splitting trees in isolation. A splitting tree which goes from one coupled sector c to N uncoupled sectors $a_1, a_2, ..., a_N$ needs $N-2$ additional internal sector labels $e_1, ..., e_{N-2}$, and, if `FusionStyle(I) isa GenericFusion` , $N-1$ additional multiplicity labels $\mu_1, ..., \mu_{N-1}$. We henceforth refer to them as vertex labels, as they are associated with the vertices of the splitting tree. In the case of `FusionStyle(I) isa UniqueFusion` , the internal sectors $e_1, ..., e_{N-2}$ are completely fixed, for `FusionStyle(I) isa MultipleFusion` they can also take different values. In our abstract notation of the splitting basis $X^{a_1a_2\ldots a_N}_{c,\alpha}$ used above, α can be considered a collective label, i.e. $\alpha = (e_1, ..., e_{N-2}; \mu_1, ..., \mu_{N-1})$. Indeed, we can check the orthogonality condition $(X^{a_1a_2\ldots a_N}_{c,\alpha})^\dagger \circ X^{a_1a_2\ldots a_N}_{c',\alpha'} = \delta_{c,c'}\delta_{\alpha,\alpha'}\mathrm{id}_c$, which now forces all internal lines e_k and vertex labels μ_l to be the same.

There is one subtle remark that we have so far ignored. Within the specific subtypes of `Sector` , we do not explicitly distinguish between R_a^* (simply denoted as a^* and graphically depicted as an upgoing arrow a) and $R_{\bar{a}}$ (simply denoted as $\bar{a}$ and depicted with a downgoing arrow), i.e. between the dual space of R_a on which the conjugated irrep acts, or the irrep $\bar{a}$ to which the complex conjugate of irrep a is isomorphic. This distinction is however important, when certain uncoupled sectors in the fusion tree actually originate from a dual space. We use the isomorphisms $Z_a : R_a^* \to R_{\bar{a}}$ and its adjoint $Z_a^\dagger : R_{\bar{a}} \to R_a^*$, as introduced in the section on topological data of a fusion category, to build fusion and splitting trees that take the distinction between irreps and their conjugates into account. Hence, in the previous example, if e.g. the first and third space in the codomain and the second space in the domain of the tensor were dual spaces, the actual pair of splitting and fusion tree would look as

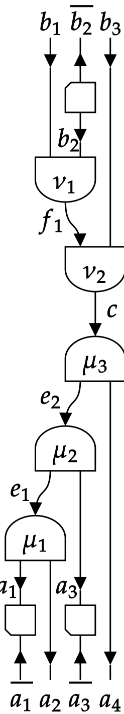
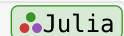


Figure 8.2: extended double fusion tree

The presence of these isomorphisms will be important when we start to bend lines, to move uncoupled sectors from the incoming to the outgoing part of the fusion-splitting tree. Note that we can still represent the fusion tree as the adjoint of a corresponding splitting tree, because we also use the adjoint of the Z isomorphisms in the splitting part, and the Z isomorphism in the fusion part. Furthermore, the presence of the Z isomorphisms does not affect the orthonormality.

We represent splitting trees and their adjoints using a specific immutable type called `FusionTree` (which actually represents a splitting tree, but fusion tree is a more common term), defined as




```
struct FusionTree{I<:Sector,N,M,L}
    uncoupled::NTuple{N,I}
    coupled::I
    isdual::NTuple{N,Bool}
    innerlines::NTuple{M,I} # fixed to M = N-2
    vertices::NTuple{L,Int} # fixed to L = N-1
end
```

Here, the fields are probably self-explanatory. The `isdual` field indicates whether an isomorphism is present (if the corresponding value is `true`) or not. Note that the field `uncoupled` contains the sectors coming out of the splitting trees, before the possible Z isomorphism, i.e. the splitting tree in the above example would have `sectors = (a1, a2, a3, a4)` . The `FusionTree` type has a number of basic properties and capabilities, such as checking for equality with `==` and support for `hash(f::FusionTree, h::UInt)` , as splitting and fusion trees are used as keys in look-up tables (i.e. `AbstractDictionary` instances) to look up certain parts of the data of a tensor.

`FusionTree` instances are not checked for consistency (i.e. valid fusion rules etc) upon creation, hence, they are assumed to be created correctly. The most natural way to create them is by using the `fusiontrees(uncoupled::NTuple{N, I}, coupled::I = unit(I))` method, which returns an iterator over all possible fusion trees from a set of `N` uncoupled sectors to a given coupled sector, which

by default is assumed to be the trivial sector of that group or fusion category (i.e. the identity object in categorical nomenclature). The return type of `fusiontrees` is a custom type `FusionTreeIterator` which conforms to the complete interface of an iterator, and has a custom `length` function that computes the number of possible fusion trees without iterating over all of them explicitly. This is best illustrated with some examples

```julia
julia> s = Irrep[SU₂](1/2)
Irrep[SU₂](1/2)

julia> collect(fusiontrees((s, s, s, s)))
2-element Vector{FusionTree{SU2Irrep, 4, 2, 3}}:
 FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 0, (false, false, false, false), (0,
 1/2))
 FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 0, (false, false, false, false), (1,
 1/2))

julia> collect(fusiontrees((s, s, s, s, s), s, (true, false, false, true, false)))
5-element Vector{FusionTree{SU2Irrep, 5, 3, 4}}:
 FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2, 1/2), 1/2, (true, false, false, true,
 false), (0, 1/2, 0))
 FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2, 1/2), 1/2, (true, false, false, true,
 false), (1, 1/2, 0))
 FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2, 1/2), 1/2, (true, false, false, true,
 false), (0, 1/2, 1))
 FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2, 1/2), 1/2, (true, false, false, true,
 false), (1, 1/2, 1))
 FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2, 1/2), 1/2, (true, false, false, true,
 false), (1, 3/2, 1))

julia> iter = fusiontrees(ntuple(n -> s, 16))
TensorKit.FusionTreeIterator{SU2Irrep, 16, NTuple{16, Tuple{SU2Irrep}}}
(((Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),),
(Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),),
(Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),),
(Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),)),
Irrep[SU₂](0), (false, false, false, false, false, false, false, false, false,
false, false, false, false, false, false, false))

julia> sum(n -> 1, iter)
1430

julia> length(iter)
1430

julia> @elapsed sum(n -> 1, iter)
0.026087834

julia> @elapsed length(iter)
```

```
2.8583e-5

julia> s2 = s ⊠ s
Irrep[SU₂ × SU₂](1/2, 1/2)

julia> collect(fusiontrees((s2, s2, s2, s2)))
4-element Vector{FusionTree{ProductSector{Tuple{SU2Irrep, SU2Irrep}}, 4, 2, 3}}:
 FusionTree{Irrep[SU₂ × SU₂]}(((1/2, 1/2), (1/2, 1/2), (1/2, 1/2), (1/2, 1/2)),
 (0, 0), (false, false, false, false), ((0, 0), (1/2, 1/2)))
 FusionTree{Irrep[SU₂ × SU₂]}(((1/2, 1/2), (1/2, 1/2), (1/2, 1/2), (1/2, 1/2)),
 (0, 0), (false, false, false, false), ((1, 0), (1/2, 1/2)))
 FusionTree{Irrep[SU₂ × SU₂]}(((1/2, 1/2), (1/2, 1/2), (1/2, 1/2), (1/2, 1/2)),
 (0, 0), (false, false, false, false), ((0, 1), (1/2, 1/2)))
 FusionTree{Irrep[SU₂ × SU₂]}(((1/2, 1/2), (1/2, 1/2), (1/2, 1/2), (1/2, 1/2)),
 (0, 0), (false, false, false, false), ((1, 1), (1/2, 1/2)))
```

Note that `FusionTree` instances are shown (printed) in a way that is valid code to reproduce them, a property which also holds for both instances of `Sector` and instances of `VectorSpace` . All of those should be displayed in a way that can be copy pasted as valid code. Furthermore, we use context to determine how to print e.g. a sector. In isolation, `s2` is printed as `(Irrep[SU₂](1/2) ⊠ Irrep[SU₂](1/2))` , however, within the fusion tree, it is simply printed as `(1/2, 1/2)` , because it will be converted back into a `ProductSector` , namely `Irrep[SU₂] ⊠ Irrep[SU₂]` by the constructor of `FusionTree{Irrep[SU₂] ⊠ Irrep[SU₂]}` .

8.2 Manipulations on a fusion tree

We now discuss elementary manipulations that we want to perform on or between fusion trees (where we actually mean splitting trees), which will form the building block for more general manipulations on a pair of a fusion and splitting tree discussed in the next subsection, and then for casting a general index manipulation of a tensor map as a linear operation in the basis of canonically ordered splitting and fusion trees. In this section, we will ignore the Z isomorphisms, as they are just trivially reshuffled under the different operations that we describe. These manipulations are used as low-level methods by the `TensorMap` methods discussed on the next page. As such, they are not exported by TensorKit.jl, nor do they overload similarly named methods from Julia Base (see `split` and `merge` below).

The first operation we discuss is an elementary braid of two neighbouring sectors (indices), i.e. a so-called Artin braid or Artin generator of the braid group. Because these two sectors do not appear on the same fusion vertex, some recoupling is necessary. The following represents two different ways to compute the result of such a braid as a linear combination of new fusion trees in canonical order:

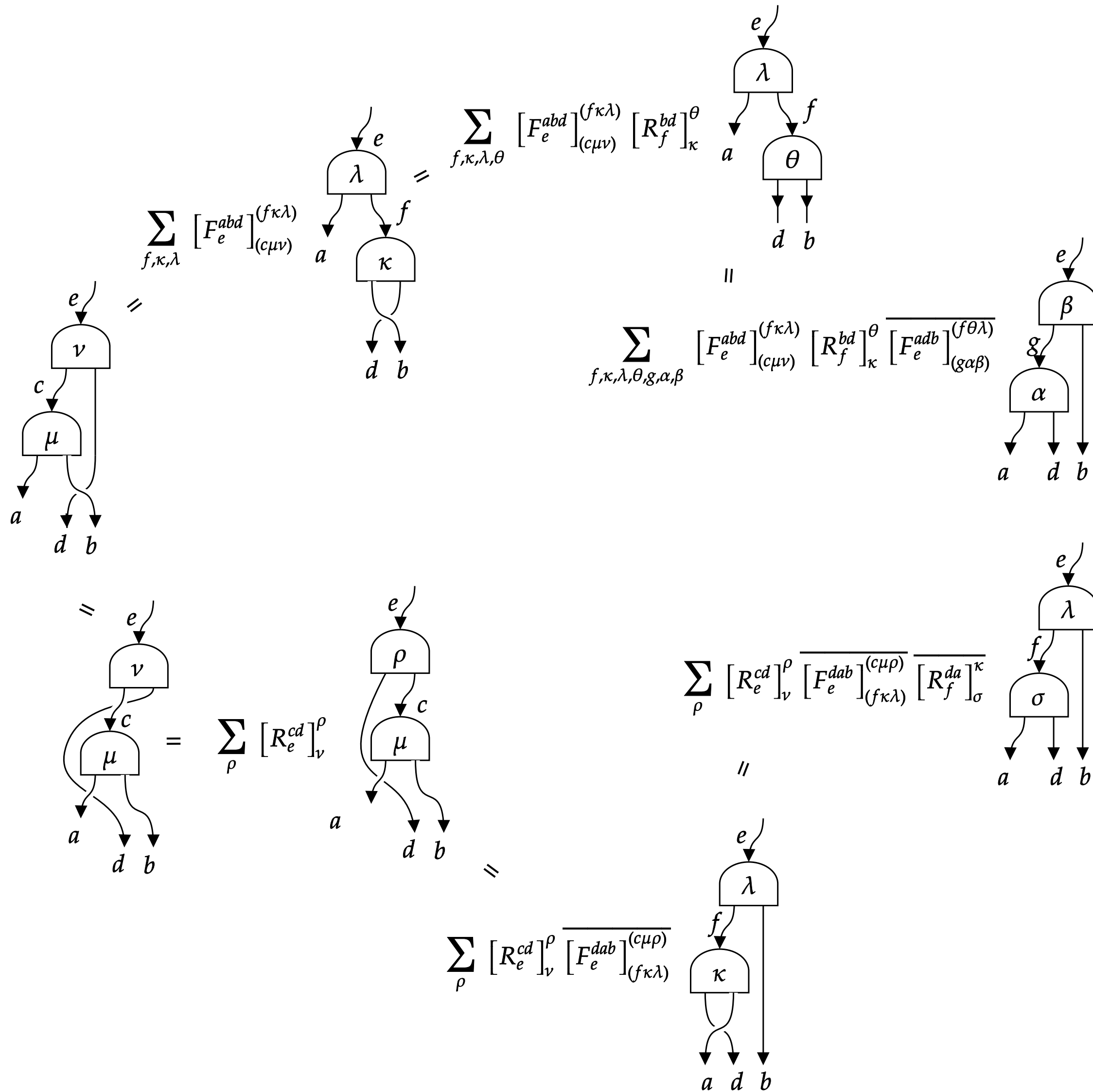


Figure 8.3: artin braid

While the upper path is the most intuitive, it requires two recouplings or F-moves (one forward and one reverse). On the other hand, the lower path requires only one (reverse) F- move, and two R-moves. The latter are less expensive to compute, and so the lower path is computationally more efficient. However, the end result should be the same, provided the pentagon and hexagon equations are satisfied. We always assume that these are satisfied for any new subtype of `Sector` , and it is up to the user to verify that they are when implementing new custom `Sector` types. This result is implemented in the function `artin_braid(f::FusionTree, i; inv = false)` where `i` denotes the position of the first sector (i.e. labeled `b` in the above graph) which is then braided with the sector at position `i+1` in the fusion tree `f` . The keyword argument `inv` allows to select the inverse braiding operation, which amounts to replacing the R-matrix with its inverse (or thus, adjoint) in the above steps. For `FusionStyle(I) isa UniqueFusion` , both `artin_braid` and `braid` return a single `Pair{FusionTree, <:Number}` , since there is exactly one output tree with a scalar (phase) coefficient. For `MultipleFusion` or `GenericFusion` sectors, use the `FusionTreeBlock` -based overloads described below.

With the elementary `artin_braid` , we can then compute a more general braid. For this, we provide an interface

```
braid(f::FusionTree{I, N}, p::IndexTuple{N}, levels::IndexTuple{N})
```

where the braid is specified as a permutation, such that the new sector at position `i` was originally at position `p[i]` , and where every uncoupled sector is also assigned a level or depth. The permutation is decomposed into swaps between neighbouring sectors, and when two sectors are swapped, their respective level will determine whether the left sector is braided over or under its right neighbor. This interface does not allow to specify the most general braid, and in particular will never wind one line around another, but can be used as a more general building block for arbitrary braids than the elementary Artin generators. A graphical example makes this probably more clear, i.e for `levels = (1, 2, 3, 4, 5)` and `permutation = (5, 3, 1, 4, 2)` , the corresponding braid is given by

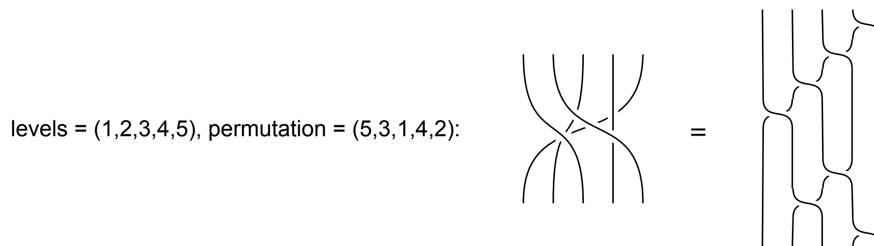


Figure 8.4: braid interface

that is, the first sector or space goes to position 3, and crosses over all other lines, because it has the lowest level (i.e. think of level as depth in the third dimension), and so forth. We sketch this operation both as a general braid on the left hand side, and as a particular composition of Artin braids on the right hand side.

When `BraidingStyle(I) == SymmetricBraiding()` , there is no distinction between applying the braiding or its inverse (i.e. lines crossing over or under each other in the graphical notation) and the whole operation simplifies down to a permutation. We then also support the interface

```
permute(f::FusionTree{I, N}, permutation::IndexTuple{N})
```

Other manipulations which are sometimes needed are

- `insertat(f1::FusionTree{I, N₁}, i::Int, f2::FusionTree{I, N₂})` : inserts a fusion tree `f2` at the `i` th uncoupled sector of fusion tree `f1` (this requires that the coupled sector `f2` matches with the `i` th uncoupled sector of `f1` , and that `!f1.isdual[i]` , i.e. that there is no Z-isomorphism on the `i` th line of `f1`), and recouple this into a linear combination of trees in canonical order, with `N₁ + N₂ - 1` uncoupled sectors, i.e. diagrammatically for `i = 3`

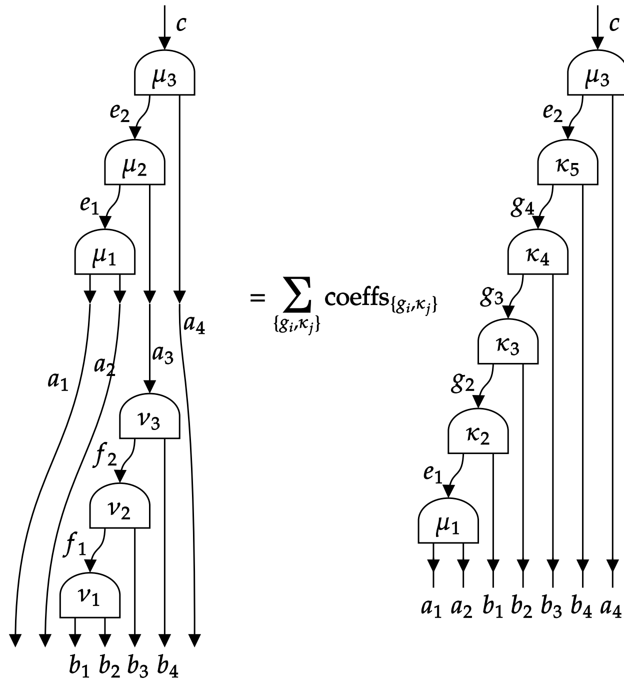


Figure 8.5: insertat

- `split(f::FusionTree{I, N}, M::Int)` : splits a fusion tree `f` into two trees `f1` and `f2` , such that `f1` has the first `M` uncoupled sectors of `f` , and `f2` the remaining `N - M` . This function is type stable if `M` is a compile time constant. `split` is the inverse of `join` : `f == join(split(f, M)...)` holds for all valid `M` . Diagrammatically, for `M = 4` , the function `split` returns

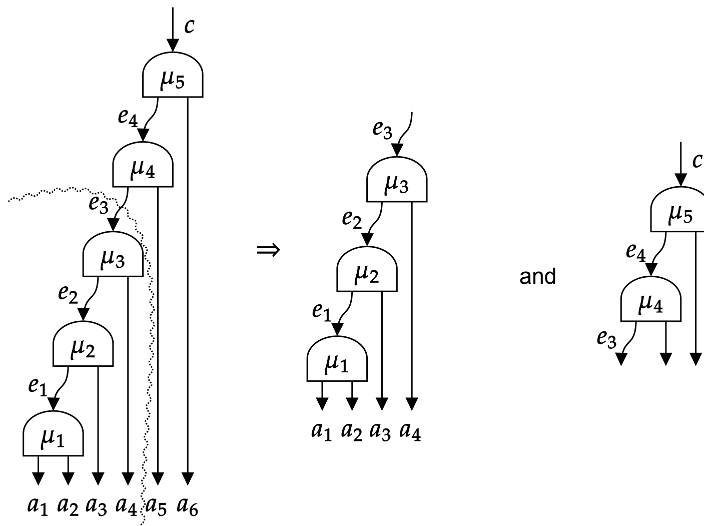


Figure 8.6: split

- `join(f1::FusionTree{I, N1}, f2::FusionTree{I, N2})` : connects the coupled sector of `f1` to the first uncoupled sector of `f2` , producing a single tree with `N1 + N2 - 1` uncoupled sectors. Requires `f1.coupled == f2.uncoupled[1]` and `!f2.isdual[1]` . This is the inverse of `split` .
- `merge(f1::FusionTree{I, N1}, f2::FusionTree{I, N2}, c::I, [μ = 1])` : merges two fusion trees `f1` and `f2` by fusing the coupled sectors of `f1` and `f2` into a sector `c` (with vertex label μ if `FusionStyle(I) == GenericFusion()`), and reexpressing the result as a linear combination of fusion trees with `N1 + N2` uncoupled sectors in canonical order. This is a simple application of `insertat` . Diagrammatically, this operation is represented as:

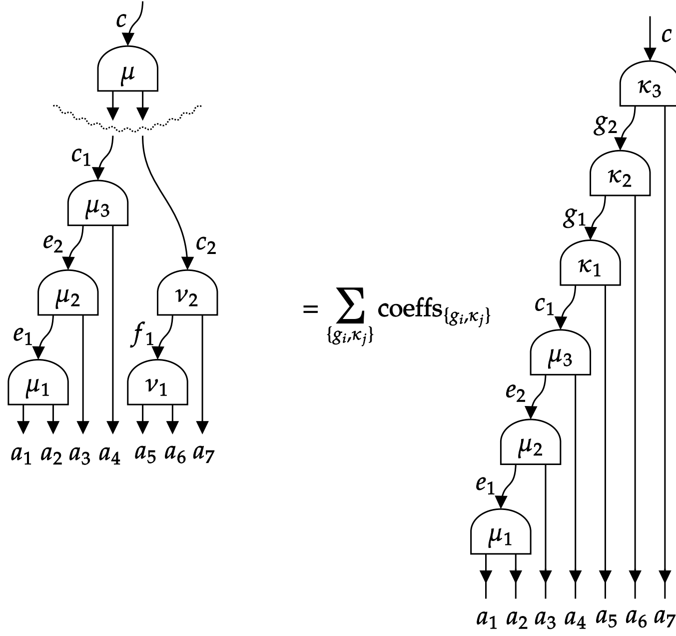


Figure 8.7: merge

8.3 Manipulations on a splitting - fusion tree pair

In this subsection we discuss manipulations that act on a splitting and fusion tree pair, which we will always represent as two separate trees `f1, f2`, where `f1` is the splitting tree and `f2` represents the fusion tree, and they should have `f1.coupled == f2.coupled`.

Two container types are used to represent such pairs, depending on the fusion style:

- `FusionTreePair{I, N₁, N₂}` is a type alias for `Tuple{FusionTree{I,N₁}, FusionTree{I,N₂}}`, i.e. a plain pair of trees sharing a coupled sector. For `FusionStyle(I) isa UniqueFusion` there is at most one tree per combination of uncoupled sectors, so a single `FusionTreePair` is the natural unit of computation. Pair manipulation functions return a `Pair{FusionTreePair, <:Number}` in this case.
- `FusionTreeBlock{I, N₁, N₂}` bundles together **all** tree pairs sharing the same uncoupled sectors. This is the natural unit for `MultipleFusion` or `GenericFusion` sectors, where multiple trees can share the same uncoupled sectors (i.e. multiple fusion channels contribute). Operations on a `FusionTreeBlock` produce a transformation matrix `U` such that the output block's basis vectors equal `U` times the input block's basis vectors, and the return type is `Pair{FusionTreeBlock, Matrix}`.

All pair manipulation functions (`repartition` , `braid` , `permute` , `transpose`) accept `Union{FusionTreePair, FusionTreeBlock}` as input.

The most important manipulation on such a pair is to move sectors from one to the other. Given the canonical order of these trees, we exclusively use the *left duality* (see the section on categories), for which the evaluation and coevaluation maps establish isomorphisms between

$$
\begin{aligned}
&\mathrm{Hom}((((b_1 \otimes b_2) \otimes \ldots) \otimes b_{N_2}), (((a_1 \otimes a_2) \otimes \ldots) \otimes a_{N_1}))\\
&\approx \mathrm{Hom}((((b_1 \otimes b_2) \otimes \ldots) \otimes b_{N_2-1}), ((((a_1 \otimes a_2) \otimes \ldots) \otimes a_{N_1}) \otimes b^*_{N_2}))\\
&\approx \mathrm{Hom}(1, (((((((a_1 \otimes a_2) \otimes \ldots) \otimes a_{N_1}) \otimes b^*_{N_2}) \otimes \ldots) \otimes b^*_2) \otimes b^*_1))
\end{aligned}
$$

where the last morphism space is then labeled by the basis of only splitting trees. We can then use the manipulations from the previous section, and then again use the left duality to bring this back to a pair of splitting and fusion tree with `N₂′` incoming and `N₁′` incoming sectors (with `N₁′ + N₂′ == N₁ + N₂`).

We now discuss how to actually bend lines, and thus, move sectors from the incoming part (fusion tree) to the outgoing part (splitting tree). Hereby, we exploit the relations between the (co)evaluation (exact pairing) and the fusion tensors, discussed in topological data of a fusion category. The main ingredient that we need is summarized in

$$= \sum_{\nu} \sqrt{\frac{d_c}{d_a}} \left[B^{ab}_c\right]^{\nu}_{\mu} \qquad \text{with} \quad \left[B^{ab}_c\right]^{\nu}_{\mu} = \sqrt{\frac{d_a d_b}{d_c}} \left[F^{ab\bar{b}}_a\right]^{(111)}_{(c\mu\nu)}$$

Figure 8.8: line bending

We will only need the B-symbol and not the A-symbol. Applying the left evaluation on the second sector of a splitting tensor thus yields a linear combination of fusion tensors (when `FusionStyle(I) == GenericFusion()`, or just a scalar times the corresponding fusion tensor otherwise), with corresponding Z ismorphism. Taking the adjoint of this relation yields the required relation to transform a fusion tensor into a splitting tensor with an added $Z^{\dagger}$ isomorphism.

However, we have to be careful if we bend a line on which a Z isomorphism (or its adjoint) is already present. Indeed, it is exactly for this operation that we explicitly need to take the presence of these isomorphisms into account. Indeed, we obtain the relation

$$= \sum_{\nu} \sqrt{\frac{d_c}{d_a}} \left[B^{ab}_c\right]^{\nu}_{\mu} = \overline{\chi_{\bar{b}}} \sum_{\nu} \sqrt{\frac{d_c}{d_a}} \left[B^{ab}_c\right]^{\nu}_{\mu}$$

$$Z_b \circ \left((Z_{\bar{b}})^{\dagger}\right)^{*} = Z_b \circ \left(Z^{*}_{\bar{b}}\right)^{\dagger} = Z_b \circ (\chi_{\bar{b}} Z_b)^{\dagger} = \overline{\chi_{\bar{b}}}\ \mathrm{id}_{\bar{b}}$$

Figure 8.9: dual line bending

Hence, bending an `isdual` sector from the splitting tree to the fusion tree yields an additional Frobenius-Schur factor, and of course leads to a normal sector (which is no longer `isdual` and does thus not come with a Z-isomorphism) on the fusion side. We again use the adjoint of this relation to bend an `isdual` sector from the fusion tree to the splitting tree.

The `FusionTree` interface to duality and line bending is given by

```
repartition(src::Union{FusionTreePair, FusionTreeBlock}, N::Int)
```

which takes a splitting tree `f1` with `N₁` outgoing sectors, a fusion tree `f2` with `N₂` incoming sectors, and applies line bending such that the resulting splitting and fusion trees have `N` outgoing sectors, corresponding to the first `N` sectors out of the list $(a_1, a_2, \ldots, a_{N_1}, b^*_{N_2}, \ldots, b^*_1)$ and `N₁ + N₂ - N` incoming sectors, corresponding to the dual of the last `N₁ + N₂ - N` sectors from the previous list, in reverse. This return values are correctly inferred if `N` is a compile time constant.

Graphically, for `N₁ = 4` , `N₂ = 3` , `N = 2` and some particular choice of `isdual` in both the fusion and splitting tree:

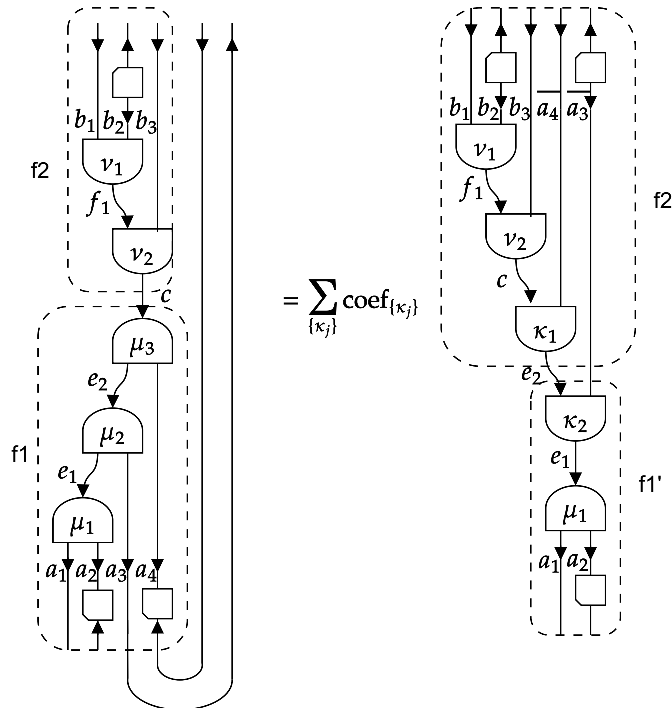


Figure 8.10: repartition

The result is returned as a `Pair` : for a `FusionTreePair` input, this is a `Pair{FusionTreePair, <:Number}` ; for a `FusionTreeBlock` input, a `Pair{FusionTreeBlock, Matrix}` . Note that the summation is only over the κ_j labels, such that, in the case of `FusionStyle(I) isa UniqueFusion` , the linear combination simplifies to a single term with a scalar coefficient.

With this basic function, we can now perform arbitrary combinations of braids or permutations with line bendings, to completely reshuffle where sectors appear. The interface provided for this is given by

```
braid(f::Union{FusionTreePair, FusionTreeBlock}, p::Index2Tuple, levels::Index2Tuple)
```

where we now have splitting tree `f1` with `N₁` outgoing sectors, a fusion tree `f2` with `N₂` incoming sectors, `levels1` and `levels2` assign a level or depth to the corresponding uncoupled sectors in `f1` and `f2` , and we represent the new configuration as a pair `p1` and `p2` . Together, `(p1..., p2...)` represents a permutation of length `N₁ + N₂ = N₁′ + N₂′` , where `p1` indicates which of the original sectors should appear as outgoing sectors in the new splitting tree and `p2` indicates which appear as incoming sectors in the new fusion tree. Hereto, we label the uncoupled sectors of `f1` from `1` to `N₁` , followed by the uncoupled sectors of `f2` from `N₁ + 1` to `N₁ + N₂` . Note that simply repartitioning the splitting and fusion tree such that e.g. all sectors appear in the new splitting tree (i.e. are outgoing), amounts to chosing `p1 = (1,..., N₁, N₁ + N₂, N₁ + N₂ - 1, ... , N₁ + 1)` and `p2 = ()` , because the duality isomorphism reverses the order of the tensor product.

This routine is implemented by indeed first making all sectors outgoing using the `repartition` function discussed above, such that only splitting trees remain, then braiding those using the routine from the previous subsection such that the new outgoing sectors appear first, followed by the new incoming sectors (in reverse order), and then again invoking the `repartition` routine to bring everything in final form. The result is again returned as a `Pair` , with the same conventions as for `repartition` .

As before, there is a simplified interface for the case where `BraidingStyle(I) isa SymmetricBraiding` and the levels are not needed. This is simply given by

```
permute(f::Union{FusionTreePair, FusionTreeBlock}, p::Index2Tuple)
```

The `braid` and `permute` routines for double fusion trees will be the main access point for corresponding manipulations on tensors. As a consequence, results from these routines are memoized via a `@cached` mechanism. The caching strategy is controlled by `CacheStyle` : for `FusionStyle(I) isa UniqueFusion` no caching is used (`NoCache()`), since the result is a single cheap scalar coefficient; for `MultipleFusion` or `GenericFusion` sectors, a global LRU cache (`GlobalLRUCache()`) is used, as computing the transformation matrix can be expensive and results are reused frequently.

This caching implies that potential inefficiencies in the fusion tree manipulations (which we nonetheless try to avoid) will not seriously affect performance of tensor manipulations.

8.4 Inspecting fusion trees as tensors

For those cases where the fusion and splitting tensors have an explicit representation as a tensor, i.e. a morphism in the category `Vect` (this essentially coincides with the case of group representations), this explicit representation can be created, which can be useful for checking purposes. Hereto, it is necessary that the *splitting tensor* $X^{ab}_{c,\mu}$, i.e. the Clebsch-Gordan coefficients of the group, are encoded via the routine `fusiontensor(a, b, c, μ = nothing)` , where the last argument is only necessary in the case of `FusionStyle(I) == GenericFusion()` . We can then convert a `FusionTree{I, N}` into an `Array` , which will yield a rank `N + 1` array where the first `N` dimensions correspond to the uncoupled sectors, and the last dimension to the coupled sector. Note that this is mostly useful for the case of `FusionStyle(I) isa MultipleFusion` groups, as in the case of abelian groups, all irreps are one-dimensional.

Some examples:

```julia
julia> s = Irrep[SU₂](1/2)
Irrep[SU₂](1/2)

julia> iter = fusiontrees((s, s, s, s), SU2Irrep(1))
TensorKit.FusionTreeIterator{SU2Irrep, 4, NTuple{4, Tuple{SU2Irrep}}}(((Irrep[SU₂]
(1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),), (Irrep[SU₂](1/2),)), Irrep[SU₂]
(1), (false, false, false, false))

julia> f = first(iter)
FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (0,
1/2))

julia> convert(Array, f)
2×2×2×2×3 Array{Float64, 5}:
[:, :, 1, 1, 1] =
  0.0       0.707107
 -0.707107  0.0

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

```
 0.0  0.0

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

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

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

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

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

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

[:, :, 1, 1, 3] =
 0.0  0.0
 0.0  0.0

[:, :, 2, 1, 3] =
 0.0  0.0
 0.0  0.0

[:, :, 1, 2, 3] =
 0.0  0.0
 0.0  0.0

[:, :, 2, 2, 3] =
  0.0       0.707107
 -0.707107  0.0

julia> LinearAlgebra.I ≈ convert(Array, FusionTree((SU2Irrep(1/2),),
SU2Irrep(1/2), (false,), ()))
true
```

```
julia> Z = adjoint(convert(Array, FusionTree((SU2Irrep(1/2),), SU2Irrep(1/2),
(true,), ())))
2×2 adjoint(::Matrix{Float64}) with eltype Float64:
 0.0  -1.0
 1.0   0.0

julia> transpose(Z) ≈ frobenius_schur_phase(SU2Irrep(1/2)) * Z
true

julia> LinearAlgebra.I ≈ convert(Array, FusionTree((Irrep[SU₂](1),), Irrep[SU₂]
(1), (false,), ()))
true

julia> Z = adjoint(convert(Array, FusionTree((Irrep[SU₂](1),), Irrep[SU₂](1),
(true,), ())))
3×3 adjoint(::Matrix{Float64}) with eltype Float64:
 0.0   0.0  1.0
 0.0  -1.0  0.0
 1.0   0.0  0.0

julia> transpose(Z) ≈ frobenius_schur_phase(Irrep[SU₂](1)) * Z
true

julia> #check orthogonality
       for f1 in iter
         for f2 in iter
           dotproduct  = dot(convert(Array, f1), convert(Array, f2))
           println("<$f1, $f2> = $dotproduct")
         end
       end
<FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (0,
1/2)), FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false,
false), (0, 1/2))> = 3.000000000000001
<FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (0,
1/2)), FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false,
false), (1, 1/2))> = -2.183074371936607e-18
<FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (0,
1/2)), FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false,
false), (1, 3/2))> = -9.801572790577901e-18
<FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (1,
1/2)), FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false,
false), (0, 1/2))> = -2.183074371936607e-18
<FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (1,
1/2)), FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false,
false), (1, 1/2))> = 2.9999999999999987
```

```
<FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (1,
1/2)), FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false,
false), (1, 3/2))> = 8.08302043869325e-17
<FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (1,
3/2)), FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false,
false), (0, 1/2))> = -9.801572790577901e-18
<FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (1,
3/2)), FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false,
false), (1, 1/2))> = 8.08302043869325e-17
<FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false, false), (1,
3/2)), FusionTree{Irrep[SU₂]}((1/2, 1/2, 1/2, 1/2), 1, (false, false, false,
false), (1, 3/2))> = 2.999999999999999
```

Note that we take the adjoint when computing `Z` , because `convert(Array, f)` assumes `f` to be splitting tree, which is built using $Z^\dagger$. Further note that the normalization (squared) of a fusion tree is given by the dimension of the coupled sector, as we are also tracing over the id_c when checking the orthogonality by computing `dot` of the corresponding tensors.

Chapter 9

Constructing tensors and the `TensorMap` type

This last page explains how to create and manipulate tensors in TensorKit.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.

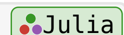
As mentioned, all tensors in TensorKit.jl are interpreted as linear maps (morphisms) from a domain (a `ProductSpace{S, N₂}`) to a codomain (another `ProductSpace{S, N₁}`), with the same `S <: ElementarySpace` that labels the type of spaces associated with the individual tensor indices. The overall type for all such tensor maps is `AbstractTensorMap{T, 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{T, S, N}` is just a synonym for `AbstractTensorMap{T, 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 $\Bbbk$.

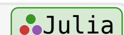
Currently, `AbstractTensorMap` has three subtypes. `TensorMap` provides the actual implementation, where the data of the tensor is stored in a `DenseArray` (more specifically a `DenseMatrix` as will be explained below). `AdjointTensorMap` is a simple wrapper type to denote the adjoint of an existing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

9.1 Storage of tensor data

Before discussion how to construct and initalize a `TensorMap` , 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

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


which can also be reshaped into matrix of size

```julia
(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(VN_1), dim(WN_2), …, 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[ℤ_2]` or `I == Irrep[U_1]`. In this case, the tensor data is associated with sectors `(a1, a2, …, aN_1) ∈ sectors(V1 ⊗ V2 ⊗ … ⊗ VN_1)` and `(b1, …, bN_2) ∈ sectors(W1 ⊗ … ⊗ WN_2)` such that they fuse to a same common charge, i.e. `(c = first(⊗(a1, …, aN_1))) == first(⊗(b1, …, bN_2))`. The data associated with this takes the form of a multidimensional array with size `(dim(V1, a1), …, dim(VN_1, aN_1), dim(W1, b1), …, dim(WN_2, bN_2))`, or equivalently, a matrix of with row size `dim(V1, a1) * … * dim(VN_1, aN_1) == dim(codomain, (a1, …, aN_1))` and column size `dim(W1, b1) * … * dim(WN_2, aN_2) == dim(domain, (b1, …, bN_2))`.

However, there are multiple combinations of `(a1, …, aN_1)` giving rise to the same `c`, and so there is data associated with all of these, as well as all possible combinations of `(b1, …, bN_2)`. 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 large matrices, which correspond to the 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 concatenate these blocks as consecutive entries in a single larger `DenseVector`, together with metadata to retrieve a block by using 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), m_1:m_2, n_1:n_2)` where the ranges `m_1:m_2` associated with `(a1, …, aN_1)` and `n_1:n_2` associated with `(b_1, …, bN_2)` 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 concatenated in a `DenseVector` , where each individual block is 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 $g \in I$, 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 this 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

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

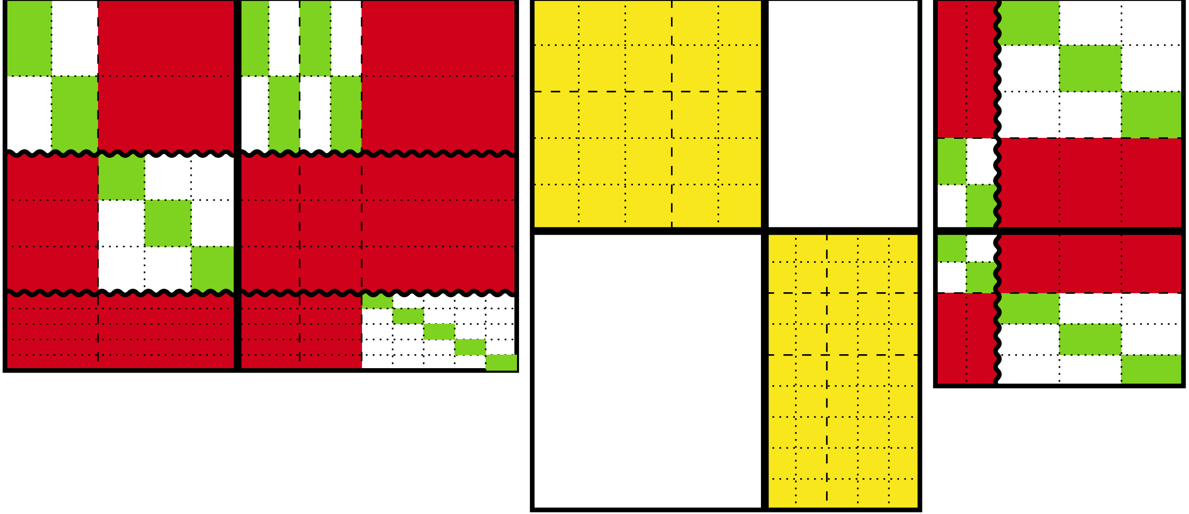


Figure 9.1: tensor storage

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, red and 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 id_c. Instead of B_c, a more accurate labelling is $t^c_{(a_1\ldots a_{N_1})\alpha,(b_1\ldots b_{N_2})\beta}$ 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 transformation, e.g. on the left (codomain) side. In more detail, it is given by

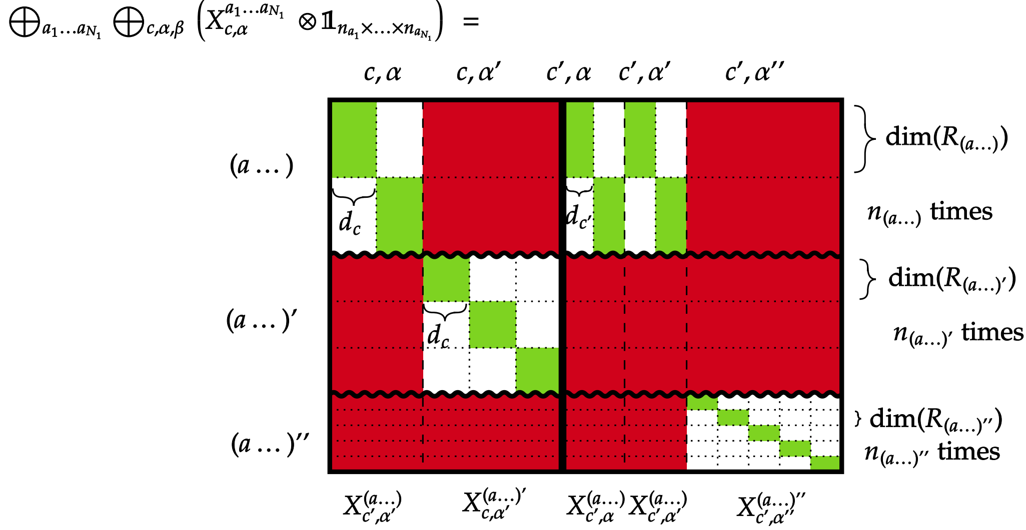


Figure 9.2: tensor unitary

Indeed, remembering that $V_i = \bigoplus_{a_i} R_{a_i} \otimes \mathbb{C}^{n_{a_i}}$ with R_{a_i} the representation space on which irrep a_i acts (with dimension $\dim(a_i)$), we find

$$V_1 \otimes \ldots \otimes V_{N_1} = \bigoplus_{a_1,\ldots,a_{N_1}} (R_{a_1} \otimes \ldots \otimes R_{a_{N_1}}) \otimes \mathbb{C}^{n_{a_1} \times \ldots n_{a_{N_1}}}.$$

In the diagram above, the wiggly 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 $(R_{a_1} \otimes \ldots \otimes R_{a_{N_1}}) \otimes \mathbb{C}^{n_{a_1} \times \ldots n_{a_{N_1}}}$ 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 $X^{(a\ldots)}_{c,\alpha} : R_{(a\ldots)} \to R_c$, the action is $X^{(a\ldots)}_{c,\alpha} \otimes \mathbb{1}_{n_{(a\ldots)}}$, 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 transformations 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))` .

9.2 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

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

Here, `f` is any of the typical functions from Base that normally create arrays, namely `zeros` , `ones` , `rand` , `randn` and `Random.randexp` . Remember that `one(codomain)` is the empty `ProductSpace{S, 0}()` . The third and fourth calling syntax use the `UndefInitializer` from Julia Base and generates a `TensorMap` with unitialized data, which can 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
julia> t1 = randn(ℂ^2 ⊗ ℂ^3, ℂ^2)
2×3←2 TensorMap{Float64, ComplexSpace, 2, 1, Vector{Float64}}:
 codomain: (ℂ^2 ⊗ ℂ^3)
 domain: ⊗(ℂ^2)
 blocks:
 * Trivial() => 6×2 reshape(view(::Vector{Float64}, 1:12), 6, 2) with eltype
Float64:
  0.00914201  -0.0919441
  1.0668      -0.427304
  1.64966      0.700277
```

```
  1.17695     -0.476235
  2.00677      0.689054
 -0.36443      0.514741

julia> t2 = zeros(Float32, ℂ^2 ⊗ ℂ^3 ← ℂ^2)
2×3←2 TensorMap{Float32, ComplexSpace, 2, 1, Vector{Float32}}:
 codomain: (ℂ^2 ⊗ ℂ^3)
 domain: ⊗(ℂ^2)
 blocks:
 * Trivial() => 6×2 reshape(view(::Vector{Float32}, 1:12), 6, 2) with eltype
 Float32:
 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> t3 = TensorMap{Float64}(undef, ℂ^2 → ℂ^2 ⊗ ℂ^3)
2×3←2 TensorMap{Float64, ComplexSpace, 2, 1, Vector{Float64}}:
 codomain: (ℂ^2 ⊗ ℂ^3)
 domain: ⊗(ℂ^2)
 blocks:
 * Trivial() => 6×2 reshape(view(::Vector{Float64}, 1:12), 6, 2) with eltype
 Float64:
 2.37414e-314  2.37414e-314
 2.37414e-314  2.37414e-314
 2.37414e-314  2.37414e-314
 2.37414e-314  2.37414e-314
 2.37414e-314  2.37414e-314
 2.37414e-314  2.37414e-314

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; digits = 3));

julia> t1[] |> disp
2×3×2 StridedViews.StridedView{Float64, 3, GenericMemory{:not_atomic, Float64,
Core.AddrSpace{Core}(0x00)}, typeof(identity)}:
[:, :, 1] =
```

```
 0.009  1.649   2.006
 1.066  1.176  -0.364

[:, :, 2] =
 -0.091   0.7    0.689
 -0.427  -0.476  0.514

julia> block(t1, Trivial()) |> disp
6×2 Array{Float64, 2}:
  0.009  -0.091
  1.066  -0.427
  1.649   0.7
  1.176  -0.476
  2.006   0.689
 -0.364   0.514

julia> reshape(t1[], dim(codomain(t1)), dim(domain(t1))) |> disp
6×2 Array{Float64, 2}:
  0.009  -0.091
  1.066  -0.427
  1.649   0.7
  1.176  -0.476
  2.006   0.689
 -0.364   0.514
```

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` . Ongoing work extends this to support for `CuMatrix` from CuArrays.jl to harness GPU computing power, and future work might include distributed arrays.

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

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 `AbstractDict` 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))` .

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. This procedure can be sketched using a simple physical example, namely the SWAP gate on two qubits,

$$\begin{aligned} \mathrm{SWAP} : \mathbb{C}^2 \otimes \mathbb{C}^2 &\to \mathbb{C}^2 \otimes \mathbb{C}^2 \\ |\, i\rangle \otimes |\, j\rangle &\mapsto |\, j\rangle \otimes |\, i\rangle. \end{aligned}$$

This operator can be rewritten in terms of the familiar Heisenberg exchange interaction $\vec{S}_i \cdot \vec{S}_j$ as

$$\mathrm{SWAP} = 2\vec{S}_i \cdot \vec{S}_j + \frac{1}{2}\,\mathbb{1},$$

where $\vec{S} = (S^x, S^y, S^z)$ and the spin-1/2 generators of SU_2 S^k are defined defined in terms of the 2×2 Pauli matrices σ^k as $S^k = \frac{1}{2}\sigma^k$. The SWAP gate can be realized as a rank-4 `TensorMap` in the following way:

```julia
julia> # encode the matrix elements of the swap gate into a rank-4 array, where the first two
       # indices correspond to the codomain and the last two indices correspond to the domain
       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> # the swap gate then maps the last two indices on the first two in reversed order
       data[1,1,1,1] = data[2,2,2,2] = data[1,2,2,1] = data[2,1,1,2] = 1
1
```

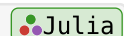
```
julia> V1 = ℂ^2 # generic qubit hilbert space
ℂ^2

julia> t1 = TensorMap(data, V1 ⊗ V1, V1 ⊗ V1)
2×2←2×2 TensorMap{Float64, ComplexSpace, 2, 2, Vector{Float64}}:
 codomain: (ℂ^2 ⊗ ℂ^2)
 domain: (ℂ^2 ⊗ ℂ^2)
 blocks:
 * Trivial() => 4×4 reshape(view(::Vector{Float64}, 1:16), 4, 4) with eltype
 Float64:
 1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  0.0  1.0

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

julia> t2 = TensorMap(data, V2 ⊗ V2, V2 ⊗ V2)
2×2←2×2 TensorMap{Float64, Rep[SU₂], 2, 2, Vector{Float64}}:
 codomain: (Rep[SU₂](1/2 => 1) ⊗ Rep[SU₂](1/2 => 1))
 domain: (Rep[SU₂](1/2 => 1) ⊗ Rep[SU₂](1/2 => 1))
 blocks:
 * Irrep[SU₂](0) => 1×1 reshape(view(::Vector{Float64}, 1:1), 1, 1) with eltype
 Float64:
 -1.0000000000000002

 * Irrep[SU₂](1) => 1×1 reshape(view(::Vector{Float64}, 2:2), 1, 1) with eltype
 Float64:
 0.9999999999999998

julia> V3 = U1Space(1/2=>1,-1/2=>1) # restricted space that only uses the `σ_z`
rotation symmetry
Rep[U₁](…) of dim 2:
  1/2 => 1
 -1/2 => 1

julia> t3 = TensorMap(data, V3 ⊗ V3, V3 ⊗ V3)
2×2←2×2 TensorMap{Float64, Rep[U₁], 2, 2, Vector{Float64}}:
 codomain: (Rep[U₁](1/2 => 1, -1/2 => 1) ⊗ Rep[U₁](1/2 => 1, -1/2 => 1))
 domain: (Rep[U₁](1/2 => 1, -1/2 => 1) ⊗ Rep[U₁](1/2 => 1, -1/2 => 1))
 blocks:
 * Irrep[U₁](0) => 2×2 reshape(view(::Vector{Float64}, 1:4), 2, 2) with eltype
 Float64:
```

```
 0.0  1.0
 1.0  0.0

 * Irrep[U₁](1) => 1×1 reshape(view(::Vector{Float64}, 5:5), 1, 1) with eltype
 Float64:
 1.0

 * Irrep[U₁](-1) => 1×1 reshape(view(::Vector{Float64}, 6:6), 1, 1) with eltype
 Float64:
 1.0

julia> for (c,b) in blocks(t3)
           println("Data for block $c :")
           disp(b)
           println()
       end
Data for block Irrep[U₁](0) :
2×2 Array{Float64, 2}:
 0.0  1.0
 1.0  0.0
Data for block Irrep[U₁](1) :
1×1 Array{Float64, 2}:
 1.0
Data for block Irrep[U₁](-1) :
1×1 Array{Float64, 2}:
 1.0
```

Hence, we recognize that the exchange interaction has eigenvalue -1 in the coupled spin zero sector (`SU2Irrep(0)`), and eigenvalue $+1$ 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 $+1$. The coupled charge `U1Irrep(0)` sector is two-dimensional, and has an eigenvalue $+1$ and an eigenvalue -1.

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 facilitate 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](…) of dim 14:
 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.

Assigning block data after initialization

In order to avoid having to know the internal structure of each representation space to properly construct the full `data` array, it is often simpler to assign the block data directly after initializing an all zero `TensorMap` with the correct spaces. While this may seem more difficult at first sight since it requires knowing the exact entries associated to each valid combination of domain uncoupled sectors, coupled sector and codomain uncoupled sectors, this is often a far more natural procedure in practice.

A first option is to directly set the full matrix block for each coupled sector in the `TensorMap` . For the example with U_1 symmetry, this can be done as

```julia
julia> t4 = zeros(V3 ⊗ V3, V3 ⊗ V3);

julia> block(t4, U1Irrep(0)) .= [1 0; 0 1];

julia> block(t4, U1Irrep(1)) .= [1;;];

julia> block(t4, U1Irrep(-1)) .= [1;;];

julia> for (c, b) in blocks(t4)
           println("Data for block $c :")
           disp(b)
           println()
       end
Data for block Irrep[U₁](0) :
2×2 Array{Float64, 2}:
 1.0  0.0
 0.0  1.0
Data for block Irrep[U₁](1) :
1×1 Array{Float64, 2}:
 1.0
Data for block Irrep[U₁](-1) :
1×1 Array{Float64, 2}:
 1.0
```

While this indeed does not require considering the internal structure of the representation spaces, it still requires knowing the precise row and column indices corresponding to each set of uncoupled sectors in the codmain and domain respectively to correctly assign the nonzero entries in each block.

Perhaps the most natural way of constructing a particular `TensorMap` is to directly assign the data slices for each splitting - fusion tree pair using the `fusiontrees(::TensorMap)` method. This returns an iterator over all tuples `(f₁, f₂)` of splitting - fusion tree pairs corresponding to all ways in which the set of domain uncoupled sectors can fuse to a coupled sector and split back into the set of codomain uncoupled sectors. By directly setting the corresponding data slice `t[f₁, f₂]` of size `(dims(codomain(t), f₁.uncoupled)..., dims(domain(t), f₂.uncoupled)...)`, we can construct all the block data without worrying about the internal ordering of row and column indices in each block. In addition, the corresponding value of each fusion tree slice is often directly informed by the object we are trying to construct in the first place. For example, in order to construct the Heisenberg exchange interaction on two spin-1/2 particles i and j as an SU_2 symmetric `TensorMap`, we can make use of the observation that

$$\vec{S}_i \cdot \vec{S}_j = \frac{1}{2}\left(\left(\vec{S}_i \cdot \vec{S}_j\right)^2 - \vec{S}_i^2 - \vec{S}_j^2\right).$$

Recalling some basic group theory, we know that the quadratic Casimir of SU_2, $\vec{S}^2$, has a well-defined eigenvalue $j(j+1)$ on every irrep of spin j. From the above expressions, we can therefore directly read off the eigenvalues of the SWAP gate in terms of this Casimir eigenvalue on the domain uncoupled sectors and the coupled sector. This gives us exactly the prescription we need to assign the data slice corresponding to each splitting - fusion tree pair:

```julia
julia> C(s::SU2Irrep) = s.j * (s.j + 1)
C (generic function with 1 method)

julia> t5 = zeros(V2 ⊗ V2, V2 ⊗ V2);

julia> for (f₁, f₂) in fusiontrees(t5)
           t5[f₁, f₂] .= C(f₂.coupled) - C(f₂.uncoupled[1]) - C(f₂.uncoupled[2]) +
           1/2
       end

julia> for (c, b) in blocks(t5)
           println("Data for block $c :")
           disp(b)
           println()
       end
Data for block Irrep[SU₂](0) :
1×1 Array{Float64, 2}:
 -1.0
Data for block Irrep[SU₂](1) :
1×1 Array{Float64, 2}:
 1.0
```

Constructing similar tensors

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

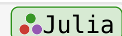
```julia
similar(t [, T::Type{<:Number}, codomain, domain])
```

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 `DenseVector` (e.g. `Vector` or `CuVector` or …) as `t` . In particular, this uses the methods `storagetype(t)` and `TensorKit.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 multiplication 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{<:DenseVector}, V)` to specify the type of `DenseVector` (and its `eltype`), e.g. `A = Vector{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{<:DenseVector}, ] 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 `DenseVector` that is currently used to store the rather trivial data of this tensor. If `InnerProductStyle(u) <: EuclideanProduct` , the same result can be obtained with the method `u = unitary([A::Type{<:DenseVector}, ] codomain, domain)` . Note that reversing the domain and codomain yields the inverse morphism, which in the case of `EuclideanProduct` 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 `ElementarySpace` with `EuclideanProduct` , 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
julia> V1 = ℤ₂Space(0 => 3, 1 => 2)
Rep[ℤ₂](…) of dim 5:
 0 => 3
 1 => 2

julia> V2 = ℤ₂Space(0 => 2, 1 => 1)
Rep[ℤ₂](…) of dim 3:
 0 => 2
 1 => 1

julia> # First a `TensorMap{ℤ₂Space, 1, 1}`
       m = randn(V1, V2)
```

```
5←3 TensorMap{Float64, Rep[ℤ₂], 1, 1, Vector{Float64}}:
 codomain: ⊗(Rep[ℤ₂](0 => 3, 1 => 2))
 domain: ⊗(Rep[ℤ₂](0 => 2, 1 => 1))
 blocks:
 * Irrep[ℤ₂](0) => 3×2 reshape(view(::Vector{Float64}, 1:6), 3, 2) with eltype
 Float64:
  1.16748    1.1334
 -1.12819   -0.53201
  0.728648   0.788414

 * Irrep[ℤ₂](1) => 2×1 reshape(view(::Vector{Float64}, 7:8), 2, 1) with eltype
 Float64:
 1.6281941444613257
 0.19547988901088562

julia> convert(Array, m) |> disp
5×3 Array{Float64, 2}:
  1.167   1.133  0.0
 -1.128  -0.532  0.0
  0.728   0.788  0.0
  0.0     0.0    1.628
  0.0     0.0    0.195

julia> # compare with:
       block(m, Irrep[ℤ₂](0)) |> disp
3×2 Array{Float64, 2}:
  1.167   1.133
 -1.128  -0.532
  0.728   0.788

julia> block(m, Irrep[ℤ₂](1)) |> disp
2×1 Array{Float64, 2}:
 1.628
 0.195

julia> # Now a `TensorMap{ℤ₂Space, 2, 2}`
       t = randn(V1 ⊗ V1, V2 ⊗ V2')
5×5←3×3 TensorMap{Float64, Rep[ℤ₂], 2, 2, Vector{Float64}}:
 codomain: (Rep[ℤ₂](0 => 3, 1 => 2) ⊗ Rep[ℤ₂](0 => 3, 1 => 2))
 domain: (Rep[ℤ₂](0 => 2, 1 => 1) ⊗ Rep[ℤ₂](0 => 2, 1 => 1)')
 blocks:
 * Irrep[ℤ₂](0) => 13×5 reshape(view(::Vector{Float64}, 1:65), 13, 5) with eltype
 Float64:
  0.453368  -0.36509   -0.683594   1.73166    0.284997
  1.00242    0.19601   -0.881492  -0.367113   0.422701
```

```
  0.542309   0.496301   0.1003     1.77774    0.056167
  ⋮
  0.196192  -0.36292   -0.98662   -0.100292  -0.28495
 -0.167651   0.14713   -1.09288    1.45865    0.639239
  0.520567  -0.127573  -1.1529    -0.213649  -0.269923

 * Irrep[ℤ₂](1) => 12×4 reshape(view(::Vector{Float64}, 66:113), 12, 4) with
 eltype Float64:
  0.381478  -0.331662  -1.1346     0.642607
  1.51724    2.3845     2.10032   -0.655085
  0.369794   1.52553    0.891028  -0.452767
  ⋮
  0.990786   0.14583   -0.769887  -0.457777
 -0.125448   3.23102    1.18956   -1.00633
  0.435528  -0.34153   -0.487184  -0.610784

julia> (array = convert(Array, t)) |> disp
5×5×3×3 Array{Float64, 4}:
[:, :, 1, 1] =
 0.453  -0.206   0.789  0.0     0.0
 1.002  -0.695   2.495  0.0     0.0
 0.542   0.342  -0.605  0.0     0.0
 0.0     0.0     0.0    0.046  -0.167
 0.0     0.0     0.0    0.196   0.52

[:, :, 2, 1] =
 -0.365   0.704   0.614   0.0     0.0
  0.196  -0.886  -0.777   0.0     0.0
  0.496   1.574  -0.682   0.0     0.0
  0.0     0.0     0.0     1.857   0.147
  0.0     0.0     0.0    -0.362  -0.127

[:, :, 3, 1] =
 0.0     0.0     0.0     0.372   0.99
 0.0     0.0     0.0    -0.648  -0.125
 0.0     0.0     0.0     2.25    0.435
 0.381   0.369  -0.445   0.0     0.0
 1.517  -1.6     1.453   0.0     0.0

[:, :, 1, 2] =
 -0.683   0.424  0.487   0.0     0.0
 -0.881   1.282  2.252   0.0     0.0
  0.1    -0.698  1.498   0.0     0.0
  0.0     0.0    0.0     0.299  -1.092
  0.0     0.0    0.0    -0.986  -1.152
```

```
[:, :, 2, 2] =
  1.731  0.092   0.151   0.0     0.0
 -0.367  0.752   1.153   0.0     0.0
  1.777  0.559  -0.15    0.0     0.0
  0.0    0.0     0.0    -1.394   1.458
  0.0    0.0     0.0    -0.1    -0.213

[:, :, 3, 2] =
  0.0     0.0     0.0     -1.441   0.145
  0.0     0.0     0.0     -0.86    3.231
  0.0     0.0     0.0      1.575  -0.341
 -0.331   1.525  -0.337    0.0     0.0
  2.384  -0.792  -0.258    0.0     0.0

[:, :, 1, 3] =
  0.0    0.0     0.0     0.269  -0.769
  0.0    0.0     0.0     0.396   1.189
  0.0    0.0     0.0    -0.605  -0.487
 -1.134  0.891  -1.429   0.0     0.0
  2.1    0.41   -1.202   0.0     0.0

[:, :, 2, 3] =
  0.0     0.0    0.0     0.65   -0.457
  0.0     0.0    0.0     0.617  -1.006
  0.0     0.0    0.0    -1.044  -0.61
  0.642  -0.452  0.838   0.0     0.0
 -0.655   1.129  1.048   0.0     0.0

[:, :, 3, 3] =
 0.284  1.019  -0.435   0.0     0.0
 0.422  0.106  -0.684   0.0     0.0
 0.056  0.177   0.517   0.0     0.0
 0.0    0.0     0.0     1.992   0.639
 0.0    0.0     0.0    -0.284  -0.269

julia> d1 = dim(codomain(t))
25

julia> d2 = dim(domain(t))
9

julia> (matrix = reshape(array, d1, d2)) |> disp
25×9 Array{Float64, 2}:
  0.453  -0.365   0.0     -0.683   1.731   0.0     0.0     0.0     0.284
```

```
  1.002   0.196   0.0     -0.881  -0.367   0.0      0.0      0.0      0.422
  0.542   0.496   0.0      0.1     1.777   0.0      0.0      0.0      0.056
  0.0     0.0     0.381    0.0     0.0    -0.331   -1.134    0.642    0.0
  0.0     0.0     1.517    0.0     0.0     2.384    2.1     -0.655    0.0
 -0.206   0.704   0.0      0.424   0.092   0.0      0.0      0.0      1.019
 -0.695  -0.886   0.0      1.282   0.752   0.0      0.0      0.0      0.106
  0.342   1.574   0.0     -0.698   0.559   0.0      0.0      0.0      0.177
  0.0     0.0     0.369    0.0     0.0     1.525    0.891   -0.452    0.0
  0.0     0.0    -1.6      0.0     0.0    -0.792    0.41     1.129    0.0
  0.789   0.614   0.0      0.487   0.151   0.0      0.0      0.0     -0.435
  2.495  -0.777   0.0      2.252   1.153   0.0      0.0      0.0     -0.684
 -0.605  -0.682   0.0      1.498  -0.15    0.0      0.0      0.0      0.517
  0.0     0.0    -0.445    0.0     0.0    -0.337   -1.429    0.838    0.0
  0.0     0.0     1.453    0.0     0.0    -0.258   -1.202    1.048    0.0
  0.0     0.0     0.372    0.0     0.0    -1.441    0.269    0.65     0.0
  0.0     0.0    -0.648    0.0     0.0    -0.86     0.396    0.617    0.0
  0.0     0.0     2.25     0.0     0.0     1.575   -0.605   -1.044    0.0
  0.046   1.857   0.0      0.299  -1.394   0.0      0.0      0.0      1.992
  0.196  -0.362   0.0     -0.986  -0.1     0.0      0.0      0.0     -0.284
  0.0     0.0     0.99     0.0     0.0     0.145   -0.769   -0.457    0.0
  0.0     0.0    -0.125    0.0     0.0     3.231    1.189   -1.006    0.0
  0.0     0.0     0.435    0.0     0.0    -0.341   -0.487   -0.61     0.0
 -0.167   0.147   0.0     -1.092   1.458   0.0      0.0      0.0      0.639
  0.52   -0.127   0.0     -1.152  -0.213   0.0      0.0      0.0     -0.269

julia> (u = reshape(convert(Array, unitary(codomain(t), fuse(codomain(t)))), d1,
d1)) |> disp
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  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  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  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  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  0.0  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  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  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
 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  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  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  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  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  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  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  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  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  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  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  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  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  0.0  0.0  0.0  0.0
 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

julia> (v = reshape(convert(Array, unitary(domain(t), fuse(domain(t)))), d2, d2))
|> disp
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}:
  0.453  -0.365  -0.683   1.731   0.284   0.0     0.0     0.0     0.0
  1.002   0.196  -0.881  -0.367   0.422   0.0     0.0     0.0     0.0
  0.542   0.496   0.1     1.777   0.056   0.0     0.0     0.0     0.0
 -0.206   0.704   0.424   0.092   1.019   0.0     0.0     0.0     0.0
 -0.695  -0.886   1.282   0.752   0.106   0.0     0.0     0.0     0.0
  0.342   1.574  -0.698   0.559   0.177   0.0     0.0     0.0     0.0
  0.789   0.614   0.487   0.151  -0.435   0.0     0.0     0.0     0.0
  2.495  -0.777   2.252   1.153  -0.684   0.0     0.0     0.0     0.0
 -0.605  -0.682   1.498  -0.15    0.517   0.0     0.0     0.0     0.0
  0.046   1.857   0.299  -1.394   1.992   0.0     0.0     0.0     0.0
  0.196  -0.362  -0.986  -0.1    -0.284   0.0     0.0     0.0     0.0
 -0.167   0.147  -1.092   1.458   0.639   0.0     0.0     0.0     0.0
  0.52   -0.127  -1.152  -0.213  -0.269   0.0     0.0     0.0     0.0
  0.0     0.0     0.0     0.0     0.0     0.381  -0.331  -1.134   0.642
  0.0     0.0     0.0     0.0     0.0     1.517   2.384   2.1    -0.655
  0.0     0.0     0.0     0.0     0.0     0.369   1.525   0.891  -0.452
  0.0     0.0     0.0     0.0     0.0    -1.6    -0.792   0.41    1.129
  0.0     0.0     0.0     0.0     0.0    -0.445  -0.337  -1.429   0.838
  0.0     0.0     0.0     0.0     0.0     1.453  -0.258  -1.202   1.048
  0.0     0.0     0.0     0.0     0.0     0.372  -1.441   0.269   0.65
  0.0     0.0     0.0     0.0     0.0    -0.648  -0.86    0.396   0.617
  0.0     0.0     0.0     0.0     0.0     2.25    1.575  -0.605  -1.044
  0.0     0.0     0.0     0.0     0.0     0.99    0.145  -0.769  -0.457
  0.0     0.0     0.0     0.0     0.0    -0.125   3.231   1.189  -1.006
  0.0     0.0     0.0     0.0     0.0     0.435  -0.341  -0.487  -0.61

julia> # compare with:
       block(t, Z2Irrep(0)) |> disp
13×5 Array{Float64, 2}:
  0.453  -0.365  -0.683   1.731   0.284
  1.002   0.196  -0.881  -0.367   0.422
  0.542   0.496   0.1     1.777   0.056
 -0.206   0.704   0.424   0.092   1.019
 -0.695  -0.886   1.282   0.752   0.106
  0.342   1.574  -0.698   0.559   0.177
  0.789   0.614   0.487   0.151  -0.435
  2.495  -0.777   2.252   1.153  -0.684
 -0.605  -0.682   1.498  -0.15    0.517
  0.046   1.857   0.299  -1.394   1.992
  0.196  -0.362  -0.986  -0.1    -0.284
 -0.167   0.147  -1.092   1.458   0.639
  0.52   -0.127  -1.152  -0.213  -0.269
```

```
julia> block(t, Z2Irrep(1)) |> disp
12×4 Array{Float64, 2}:
  0.381  -0.331  -1.134   0.642
  1.517   2.384   2.1    -0.655
  0.369   1.525   0.891  -0.452
 -1.6    -0.792   0.41    1.129
 -0.445  -0.337  -1.429   0.838
  1.453  -0.258  -1.202   1.048
  0.372  -1.441   0.269   0.65
 -0.648  -0.86    0.396   0.617
  2.25    1.575  -0.605  -1.044
  0.99    0.145  -0.769  -0.457
 -0.125   3.231   1.189  -1.006
  0.435  -0.341  -0.487  -0.61
```

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 $\mathbf{Rep}_G$, which can be interpreted as a subcategory of $\mathbf{Vect}$). Here, we are in this case with $G = \mathbb{Z}_2$. 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
julia> V1 = SU₂Space(0 => 2, 1 => 1)
Rep[SU₂](…) of dim 5:
 0 => 2
 1 => 1

julia> V2 = SU₂Space(0 => 1, 1 => 1)
Rep[SU₂](…) of dim 4:
 0 => 1
 1 => 1

julia> # First a `TensorMap{SU₂Space, 1, 1}`
       m = randn(V1, V2)
5←4 TensorMap{Float64, Rep[SU₂], 1, 1, Vector{Float64}}:
 codomain: ⊗(Rep[SU₂](0 => 2, 1 => 1))
 domain: ⊗(Rep[SU₂](0 => 1, 1 => 1))
```

```
 blocks:
 * Irrep[SU₂](0) => 2×1 reshape(view(::Vector{Float64}, 1:2), 2, 1) with eltype
 Float64:
  1.1011746807128568
 -0.8435554114699405

 * Irrep[SU₂](1) => 1×1 reshape(view(::Vector{Float64}, 3:3), 1, 1) with eltype
 Float64:
 0.9108480163408458

julia> convert(Array, m) |> disp
5×4 Array{Float64, 2}:
  1.101  0.0   0.0   0.0
 -0.843  0.0   0.0   0.0
  0.0    0.91  0.0   0.0
  0.0    0.0   0.91  0.0
  0.0    0.0   0.0   0.91

julia> # compare with:
       block(m, Irrep[SU₂](0)) |> disp
2×1 Array{Float64, 2}:
  1.101
 -0.843

julia> block(m, Irrep[SU₂](1)) |> disp
1×1 Array{Float64, 2}:
 0.91

julia> # Now a `TensorMap{SU₂Space, 2, 2}`
       t = randn(V1 ⊗ V1, V2 ⊗ V2')
5×5←4×4 TensorMap{Float64, Rep[SU₂], 2, 2, Vector{Float64}}:
 codomain: (Rep[SU₂](0 => 2, 1 => 1) ⊗ Rep[SU₂](0 => 2, 1 => 1))
 domain: (Rep[SU₂](0 => 1, 1 => 1) ⊗ Rep[SU₂](0 => 1, 1 => 1)')
 blocks:
 * Irrep[SU₂](0) => 5×2 reshape(view(::Vector{Float64}, 1:10), 5, 2) with eltype
 Float64:
 -0.585758   1.89535
 -1.9742    -0.380165
 -0.164685   1.49072
 -0.460286   0.233763
 -0.391724   0.139957

 * Irrep[SU₂](1) => 5×3 reshape(view(::Vector{Float64}, 11:25), 5, 3) with eltype
 Float64:
  0.0745116   0.254848    0.528917
```

```
 0.509086   -0.284674    0.836316
-0.391055   -1.54666    -0.0407906
 0.333141    0.165583    0.766985
-2.29302     0.0802271  -0.324213

* Irrep[SU₂](2) => 1×1 reshape(view(::Vector{Float64}, 26:26), 1, 1) with eltype
Float64:
-0.2348763355406851

julia> (array = convert(Array, t)) |> disp
5×5×4×4 Array{Float64, 4}:
[:, :, 1, 1] =
 -0.585  -0.164   0.0    0.0     0.0
 -1.974  -0.46    0.0    0.0     0.0
  0.0     0.0     0.0    0.0    -0.226
  0.0     0.0     0.0    0.226   0.0
  0.0     0.0    -0.226  0.0     0.0

[:, :, 2, 1] =
 0.0    0.0    -0.391   0.0    0.0
 0.0    0.0     0.333   0.0    0.0
 0.074  0.509   0.0    -1.621  0.0
 0.0    0.0     1.621   0.0    0.0
 0.0    0.0     0.0     0.0    0.0

[:, :, 3, 1] =
 0.0    0.0    0.0    -0.391   0.0
 0.0    0.0    0.0     0.333   0.0
 0.0    0.0    0.0     0.0    -1.621
 0.074  0.509  0.0     0.0     0.0
 0.0    0.0    1.621   0.0     0.0

[:, :, 4, 1] =
 0.0    0.0    0.0  0.0    -0.391
 0.0    0.0    0.0  0.0     0.333
 0.0    0.0    0.0  0.0     0.0
 0.0    0.0    0.0  0.0    -1.621
 0.074  0.509  0.0  1.621   0.0

[:, :, 1, 2] =
 0.0     0.0    0.0   0.0     -1.546
 0.0     0.0    0.0   0.0      0.165
 0.0     0.0    0.0   0.0      0.0
 0.0     0.0    0.0   0.0      0.056
 0.254  -0.284  0.0  -0.056    0.0
```

```
[:, :, 2, 2] =
  1.094  0.86   0.0    -0.028   0.0
 -0.219  0.134  0.0     0.542   0.0
  0.0    0.0    0.0     0.0    -0.154
  0.374  0.591  0.0    -0.124   0.0
  0.0    0.0    0.169   0.0     0.0

[:, :, 3, 2] =
 0.0    0.0    0.0  0.0    -0.028
 0.0    0.0    0.0  0.0     0.542
 0.0    0.0    0.0  0.0     0.0
 0.0    0.0    0.0  0.0    -0.279
 0.374  0.591  0.0  0.044   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  -0.234

[:, :, 1, 3] =
  0.0    0.0    0.0     1.546   0.0
  0.0    0.0    0.0    -0.165   0.0
  0.0    0.0    0.0     0.0    -0.056
 -0.254  0.284  0.0     0.0     0.0
  0.0    0.0    0.056   0.0     0.0

[:, :, 2, 3] =
  0.0     0.0     0.028  0.0    0.0
  0.0     0.0    -0.542  0.0    0.0
 -0.374  -0.591   0.0    0.279  0.0
  0.0     0.0    -0.044  0.0    0.0
  0.0     0.0     0.0    0.0    0.0

[:, :, 3, 3] =
  1.094  0.86   0.0    0.0    0.0
 -0.219  0.134  0.0    0.0    0.0
  0.0    0.0    0.0    0.0    0.124
  0.0    0.0    0.0    0.109  0.0
  0.0    0.0    0.124  0.0    0.0

[:, :, 4, 3] =
 0.0    0.0    0.0  0.0    -0.028
```

```
 0.0    0.0    0.0  0.0     0.542
 0.0    0.0    0.0  0.0     0.0
 0.0    0.0    0.0  0.0    -0.044
 0.374  0.591  0.0  0.279   0.0

[:, :, 1, 4] =
 0.0     0.0    -1.546  0.0    0.0
 0.0     0.0     0.165  0.0    0.0
 0.254  -0.284   0.0    0.056  0.0
 0.0     0.0    -0.056  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  -0.234  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     0.028  0.0    0.0
  0.0     0.0    -0.542  0.0    0.0
 -0.374  -0.591   0.0    0.044  0.0
  0.0     0.0    -0.279  0.0    0.0
  0.0     0.0     0.0    0.0    0.0

[:, :, 4, 4] =
  1.094   0.86    0.0     0.028  0.0
 -0.219   0.134   0.0    -0.542  0.0
  0.0     0.0     0.0     0.0    0.169
 -0.374  -0.591   0.0    -0.124  0.0
  0.0     0.0    -0.154   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}:
 -0.585   0.0     0.0     0.0     0.0     1.094   0.0     0.0     0.0     0.0
 1.094   0.0     0.0     0.0     0.0     1.094
 -1.974   0.0     0.0     0.0     0.0    -0.219   0.0     0.0     0.0     0.0
 -0.219   0.0     0.0     0.0     0.0    -0.219
```

```
  0.0     0.074   0.0     0.0     0.0     0.0     0.0     0.0     0.0    -0.374
  0.0     0.0     0.254   0.0    -0.374   0.0
  0.0     0.0     0.074   0.0     0.0     0.374   0.0     0.0    -0.254   0.0
  0.0     0.0     0.0     0.0     0.0    -0.374
  0.0     0.0     0.0     0.074   0.254   0.0     0.374   0.0     0.0     0.0
  0.0     0.374   0.0     0.0     0.0     0.0
 -0.164   0.0     0.0     0.0     0.0     0.86    0.0     0.0     0.0     0.0
 0.86    0.0     0.0     0.0     0.0     0.86
 -0.46    0.0     0.0     0.0     0.0     0.134   0.0     0.0     0.0     0.0
 0.134   0.0     0.0     0.0     0.0     0.134
  0.0     0.509   0.0     0.0     0.0     0.0     0.0     0.0     0.0    -0.591
  0.0     0.0    -0.284   0.0    -0.591   0.0
  0.0     0.0     0.509   0.0     0.0     0.591   0.0     0.0     0.284   0.0
  0.0     0.0     0.0     0.0     0.0    -0.591
  0.0     0.0     0.0     0.509  -0.284   0.0     0.591   0.0     0.0     0.0
  0.0     0.591   0.0     0.0     0.0     0.0
  0.0    -0.391   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.028
  0.0     0.0    -1.546   0.0     0.028   0.0
  0.0     0.333   0.0     0.0     0.0     0.0     0.0     0.0     0.0    -0.542
  0.0     0.0     0.165   0.0    -0.542   0.0
  0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
  0.0     0.0     0.0    -0.234   0.0     0.0
  0.0     1.621   0.0     0.0     0.0     0.0     0.0     0.0     0.0    -0.044
  0.0     0.0    -0.056   0.0    -0.279   0.0
 -0.226   0.0     1.621   0.0     0.0     0.169   0.0     0.0     0.056   0.0
 0.124   0.0     0.0     0.0     0.0    -0.154
  0.0     0.0    -0.391   0.0     0.0    -0.028   0.0     0.0     1.546   0.0
  0.0     0.0     0.0     0.0     0.0     0.028
  0.0     0.0     0.333   0.0     0.0     0.542   0.0     0.0    -0.165   0.0
  0.0     0.0     0.0     0.0     0.0    -0.542
  0.0    -1.621   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.279
  0.0     0.0     0.056   0.0     0.044   0.0
  0.226   0.0     0.0     0.0     0.0    -0.124   0.0     0.0     0.0     0.0
  0.109   0.0     0.0     0.0     0.0    -0.124
  0.0     0.0     0.0     1.621  -0.056   0.0     0.044   0.0     0.0     0.0
  0.0     0.279   0.0     0.0     0.0     0.0
  0.0     0.0     0.0    -0.391  -1.546   0.0    -0.028   0.0     0.0     0.0
  0.0    -0.028   0.0     0.0     0.0     0.0
  0.0     0.0     0.0     0.333   0.165   0.0     0.542   0.0     0.0     0.0
  0.0     0.542   0.0     0.0     0.0     0.0
 -0.226   0.0    -1.621   0.0     0.0    -0.154   0.0     0.0    -0.056   0.0
 0.124   0.0     0.0     0.0     0.0     0.169
  0.0     0.0     0.0    -1.621   0.056   0.0    -0.279   0.0     0.0     0.0
  0.0    -0.044   0.0     0.0     0.0     0.0
  0.0     0.0     0.0     0.0     0.0     0.0     0.0    -0.234   0.0     0.0
  0.0     0.0     0.0     0.0     0.0     0.0

julia> (u = reshape(convert(Array, unitary(codomain(t), fuse(codomain(t)))), d1,
d1)) |> disp
```

```
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    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    0.999  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    0.0
 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
 0.0  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  0.0  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  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    0.999  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  0.0  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  0.0   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   0.0    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    0.0    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    0.0    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   -0.707  0.0    0.0    0.0  0.707  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   -0.707  0.0    0.0  0.0    0.408  0.0    0.0
 0.0  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    0.0    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    0.0    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    0.0    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    0.816  0.0    0.0
 0.0  0.0  0.0  0.0   0.0    0.0    0.0    0.0    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
 0.0  0.0  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    0.0    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   0.577  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    0.707  0.0    0.0  0.0    0.408  0.0    0.0
```

```
 0.0  0.0  0.0  0.0   0.0    0.0    0.0    0.0    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
 0.0  0.0  0.0  0.0   0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    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

julia> (v = reshape(convert(Array, unitary(domain(t), fuse(domain(t)))), d2, d2))
|> disp
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    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    0.999  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    0.999  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  0.577  0.0    0.0    0.0    0.0     0.0    0.0     0.0     0.707  0.0    0.0
 0.0    0.408   0.0    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
 0.0  0.0    0.0    0.0    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    -0.999  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    -0.707   0.0    0.0    0.0
 -0.707   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   -0.816   0.0    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
 0.0  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     0.0     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    -0.707   0.0    0.0    0.0
 0.707  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    0.408   0.0    0.0

julia> u' * u ≈ I ≈ v' * v
true

julia> (u' * matrix * v) |> disp
25×16 Array{Float64, 2}:
 -0.585   1.895   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
 0.0     0.0     0.0     0.0     0.0     0.0
```

```
-1.974  -0.38   0.0     0.0     0.0     0.0     0.0     0.0     0.0    -0.0
0.0     0.0     0.0    -0.0     0.0     0.0
-0.164   1.49   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
0.0     0.0     0.0     0.0     0.0     0.0
-0.46    0.233  0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
0.0     0.0     0.0    -0.0     0.0     0.0
-0.391   0.139  0.0     0.0     0.0     0.0    -0.0     0.0     0.0    -0.0
0.0     0.0     0.0    -0.0     0.0     0.0
 0.0     0.0    0.074   0.0     0.0     0.254   0.0     0.0     0.528   0.0
 0.0     0.0   -0.0     0.0     0.0     0.0
 0.0    -0.0    0.0     0.074   0.0     0.0     0.254   0.0     0.0     0.528
 0.0     0.0    0.0     0.0     0.0     0.0
 0.0     0.0    0.0     0.0     0.074   0.0     0.0     0.254   0.0     0.0
 0.528   0.0    0.0     0.0    -0.0     0.0
 0.0     0.0    0.509   0.0     0.0    -0.284   0.0     0.0     0.836   0.0
 0.0     0.0   -0.0     0.0     0.0     0.0
 0.0    -0.0    0.0     0.509   0.0     0.0    -0.284   0.0     0.0     0.836
 0.0     0.0    0.0    -0.0     0.0     0.0
 0.0     0.0    0.0     0.0     0.509   0.0     0.0    -0.284   0.0     0.0
 0.836   0.0    0.0     0.0    -0.0     0.0
 0.0     0.0   -0.391   0.0     0.0    -1.546   0.0     0.0    -0.04    0.0
 0.0     0.0   -0.0     0.0     0.0     0.0
 0.0    -0.0    0.0    -0.391   0.0     0.0    -1.546   0.0     0.0    -0.04
 0.0     0.0    0.0    -0.0     0.0     0.0
 0.0     0.0    0.0     0.0    -0.391   0.0     0.0    -1.546   0.0     0.0
 -0.04    0.0    0.0     0.0    -0.0     0.0
 0.0     0.0    0.333   0.0     0.0     0.165   0.0     0.0     0.766   0.0
 0.0     0.0   -0.0     0.0     0.0     0.0
 0.0    -0.0    0.0     0.333   0.0     0.0     0.165   0.0     0.0     0.766
 0.0     0.0    0.0    -0.0     0.0     0.0
 0.0     0.0    0.0     0.0     0.333   0.0     0.0     0.165   0.0     0.0
 0.766   0.0    0.0     0.0    -0.0     0.0
 0.0     0.0   -2.293   0.0     0.0     0.08    0.0     0.0    -0.324   0.0
 0.0     0.0   -0.0     0.0     0.0     0.0
 0.0    -0.0    0.0    -2.293   0.0     0.0     0.08    0.0     0.0    -0.324
 0.0     0.0    0.0    -0.0     0.0     0.0
 0.0     0.0    0.0     0.0    -2.293   0.0     0.0     0.08    0.0     0.0
 -0.324   0.0    0.0     0.0    -0.0     0.0
 0.0     0.0    0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
 0.0    -0.234  0.0     0.0     0.0     0.0
 0.0     0.0   -0.0     0.0     0.0    -0.0     0.0     0.0    -0.0     0.0
 0.0     0.0   -0.234   0.0     0.0     0.0
 0.0    -0.0    0.0    -0.0     0.0     0.0    -0.0     0.0     0.0    -0.0
 0.0     0.0    0.0    -0.234   0.0     0.0
 0.0     0.0    0.0     0.0    -0.0     0.0     0.0    -0.0     0.0     0.0
 -0.0     0.0    0.0     0.0    -0.234   0.0
 0.0     0.0    0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
 0.0     0.0    0.0     0.0     0.0    -0.234
```

```
julia> # compare with:
       block(t, SU2Irrep(0)) |> disp
5×2 Array{Float64, 2}:
 -0.585   1.895
 -1.974  -0.38
 -0.164   1.49
 -0.46    0.233
 -0.391   0.139

julia> block(t, SU2Irrep(1)) |> disp
5×3 Array{Float64, 2}:
  0.074   0.254   0.528
  0.509  -0.284   0.836
 -0.391  -1.546  -0.04
  0.333   0.165   0.766
 -2.293   0.08   -0.324

julia> block(t, SU2Irrep(2)) |> disp
1×1 Array{Float64, 2}:
 -0.234
```

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` .

9.3 Tensor properties

Given a `t::AbstractTensorMap{T, S, N₁, N₂}` , there are various methods to query its properties. The most important are clearly `codomain(t)` and `domain(t)` . For `t::AbstractTensor{S, N}` , i.e. `t::AbstractTensorMap{T, S, N, 0}` , we can use `space(t)` as synonym for `codomain(t)` . However, for a general `AbstractTensorMap` this has no meaning. However, we can 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 `TensorKit.codomainind(t)` and `TensorKit.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) `DenseVector{T}` in which the matrix blocks are stored. Note that during construction, a (one-time) warning is printed if `!(T ⊂ field(S))` . The specific `DenseVector{T}` subtype in which the tensor data is stored is obtained as `storagetype(t)` . Each of the methods `numind` , `numout` , `numin` , `TensorKit.codomainind` , `TensorKit.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
julia> typeof(t)
TensorMap{Float64, GradedSpace{SU2Irrep, TensorKit.SortedVectorDict{SU2Irrep,
Int64}}, 2, 2, Vector{Float64}}

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](…) of dim 5:
 0 => 2
 1 => 1

julia> space(t,2)
Rep[SU₂](…) of dim 5:
 0 => 2
 1 => 1

julia> space(t,3)
Rep[SU₂](…)' of dim 4:
 0 => 1
 1 => 1

julia> space(t,4)
Rep[SU₂](…) of dim 4:
 0 => 1
 1 => 1

julia> numind(t)
4

julia> numout(t)
```

```
2

julia> numin(t)
2

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

julia> sectortype(t)
SU2Irrep

julia> field(t)
ℂ

julia> eltype(t)
Float64

julia> storagetype(t)
Vector{Float64} (alias for Array{Float64, 1})

julia> blocksectors(t)
3-element Dictionaries.Indices{SU2Irrep}:
 Irrep[SU₂](0)
 Irrep[SU₂](1)
 Irrep[SU₂](2)

julia> blocks(t)
blocks(::TensorMap{Float64, Rep[SU₂], 2, 2, Vector{Float64}}):
 * Irrep[SU₂](0) => 5×2 reshape(view(::Vector{Float64}, 1:10), 5, 2) with eltype
 Float64:
 -0.585758   1.89535
 -1.9742    -0.380165
 -0.164685   1.49072
 -0.460286   0.233763
 -0.391724   0.139957

 * Irrep[SU₂](1) => 5×3 reshape(view(::Vector{Float64}, 11:25), 5, 3) with eltype
 Float64:
  0.0745116   0.254848    0.528917
  0.509086   -0.284674    0.836316
 -0.391055   -1.54666    -0.0407906
  0.333141    0.165583    0.766985
 -2.29302     0.0802271  -0.324213
```

```
 * Irrep[SU₂](2) => 1×1 reshape(view(::Vector{Float64}, 26:26), 1, 1) with eltype
 Float64:
 -0.2348763355406851

julia> block(t, first(blocksectors(t)))
5×2 reshape(view(::Vector{Float64}, 1:10), 5, 2) with eltype Float64:
 -0.585758   1.89535
 -1.9742    -0.380165
 -0.164685   1.49072
 -0.460286   0.233763
 -0.391724   0.139957

julia> fusiontrees(t)
14-element Dictionaries.Indices{Tuple{FusionTree{SU2Irrep, 2, 0, 1},
FusionTree{SU2Irrep, 2, 0, 1}}}:
 (FusionTree{Irrep[SU₂]}((0, 0), 0, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((1, 1), 0, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((0, 0), 0, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((1, 1), 0, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((1, 0), 1, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((0, 1), 1, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((1, 1), 1, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((1, 0), 1, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((0, 1), 1, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((1, 1), 1, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((1, 0), 1, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((0, 1), 1, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((1, 1), 1, (false, false), ()), FusionTree{Irrep[SU₂]}…
 (FusionTree{Irrep[SU₂]}((1, 1), 2, (false, false), ()), FusionTree{Irrep[SU₂]}…

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

julia> t[f1,f2]
2×2×1×1 StridedViews.StridedView{Float64, 4, Memory{Float64}, typeof(identity)}:
[:, :, 1, 1] =
 -0.585758  -0.164685
 -1.9742    -0.460286
```

9.4 Reading and writing tensors: `Dict` conversion

There are no custom or dedicated methods for reading, writing or storing `TensorMap` s, 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, ...

Chapter 10

Manipulating tensors

10.1 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 TensorKit.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 (typically using the same method) and never needs to perform any basis transforms.

In particular, `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. TensorKit.jl also supports (and exports) the mutating method `mul!(t, t1, t2)` . We can then also try to invert a tensor map using `inv(t)` , though this can only exist if the domain and codomain are isomorphic, which can e.g. be checked as `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 `InnerProductStyle(t) <: EuclideanProduct` .

`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` . In addition, `TensorMap` supports basic Julia methods such as `fill!` and `copy!` , as well as `copy(t)` to create a copy with independent data. Aside from basic + and * operations, TensorKit.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` .

For `S = spacetype(t)` where `InnerProductStyle(S) <: EuclideanProduct` , 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. Furthermore, there is `normalize(t)` and `normalize!(t)` to return a scaled version of `t` with unit norm. These operations should also exist for `InnerProductStyle(S) <: HasInnerProduct` , but require an interface for defining a custom inner product in these spaces. Currently, there is no concrete subtype of `HasInnerProduct` that is not an `EuclideanProduct` . In particular, `CartesianSpace` , `ComplexSpace` and `GradedSpace` all have `InnerProductStyle(S) <: EuclideanProduct` .

With tensors that have `InnerProductStyle(t) <: EuclideanProduct` 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 invisible 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 = TensorKit.adjointtensorindex(t, i)` , where the latter method is typically not necessary and hence unexported. There is also a plural `TensorKit.adjointtensorindices` to convert multiple indices at once. Note that, because the adjoint interchanges domain and codomain, we have `space(t', j) == space(t, i)'` .

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

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)` as discussed above, such that for a general homomorphism `t′` , we have `t′ == id(codomain(t′)) * t′ == t′ * id(domain(t′))` . Returning to the 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)` . Furthermore, there are a number of tensor factorizations for both endomorphisms and general homomorphism that we discuss below.

Finally, there are a number of operations that also belong in this paragraph because of their analogy to common matrix operations. 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)` . If we have two `TensorMap{T, 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{T, 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
julia> V1 = ℂ^2
ℂ^2

julia> t = randn(V1 ← V1 ⊗ V1 ⊗ V1)
2←2×2×2 TensorMap{Float64, ComplexSpace, 1, 3, Vector{Float64}}:
 codomain: ⊗(ℂ^2)
 domain: (ℂ^2 ⊗ ℂ^2 ⊗ ℂ^2)
 blocks:
 * Trivial() => 2×8 reshape(view(::Vector{Float64}, 1:16), 2, 8) with eltype
 Float64:
  2.01536    0.476951  -1.93527    0.960437  …  0.314175   0.586678  -0.832983
 -0.116833   0.201891  -0.720385  -0.943761     0.0170323  0.126376  -0.519614

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

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

```
julia> dot(t2, t)
-1.5573384027630892 + 0.9391500148369417im

julia> tr(t2' * t)
-1.5573384027630885 + 0.9391500148369414im

julia> dot(t2, t) ≈ dot(t', t2')
true

julia> dot(t2, t2)
10.50857954385725 + 0.0im

julia> norm(t2)^2
10.508579543857248

julia> t3 = copy!(similar(t, ComplexF64), t);

julia> t3 == t
true

julia> rmul!(t3, 0.8);

julia> t3 ≈ 0.8 * t
true

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

julia> t3 ≈ 0.5 * t2 + 0.8 * 1.3im * t
true

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

julia> t5 = TensorMap{Float64}(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
```

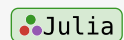
```
julia> t6 = randn(ComplexF64, V1, codomain(t));

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

julia> t7 = catcodomain(t4, t6);

julia> foreach(println, (codomain(t4), codomain(t6), codomain(t7)))
⊗(ℂ^2)
⊗(ℂ^2)
⊗(ℂ^4)

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

julia> t8 = t4 ⊗ t6;

julia> foreach(println, (codomain(t4), codomain(t6), codomain(t8)))
⊗(ℂ^2)
⊗(ℂ^2)
(ℂ^2 ⊗ ℂ^2)

julia> foreach(println, (domain(t4), domain(t6), domain(t8)))
⊗(ℂ^2)
⊗(ℂ^2)
(ℂ^2 ⊗ ℂ^2)

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

10.2 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.

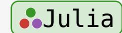
For this, we use an interface that is closely related to that for manipulating splitting- fusion tree pairs, namely `braid` and `permute` , with the interface

```
braid(t::AbstractTensorMap{T,S,N₁,N₂}, (p1, p2)::Index2Tuple{N₁′,N₂′},
levels::IndexTuple{N₁+N₂,Int})
```



and

```julia
permute(t::AbstractTensorMap{T,S,N₁,N₂}, (p1, p2)::Index2Tuple{N₁′,N₂′};
copy = false)
```



both of which return an instance of `AbstractTensorMap{T, 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 TensorKit.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, (p1, p2), levels)` and `permute!(tdst, tsrc, (p1, p2))`.

Another operation that belongs under 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 TensorKit.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.

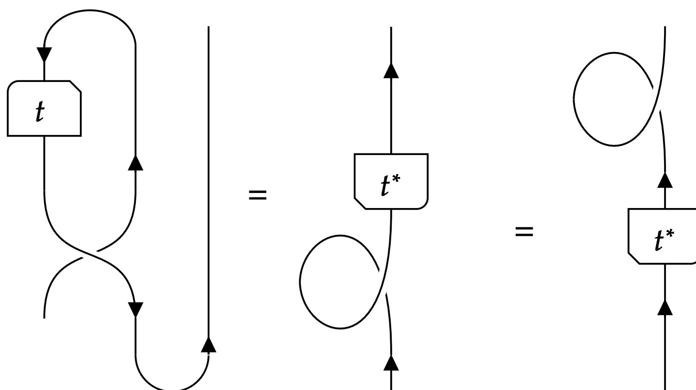


Figure 10.1: transpose

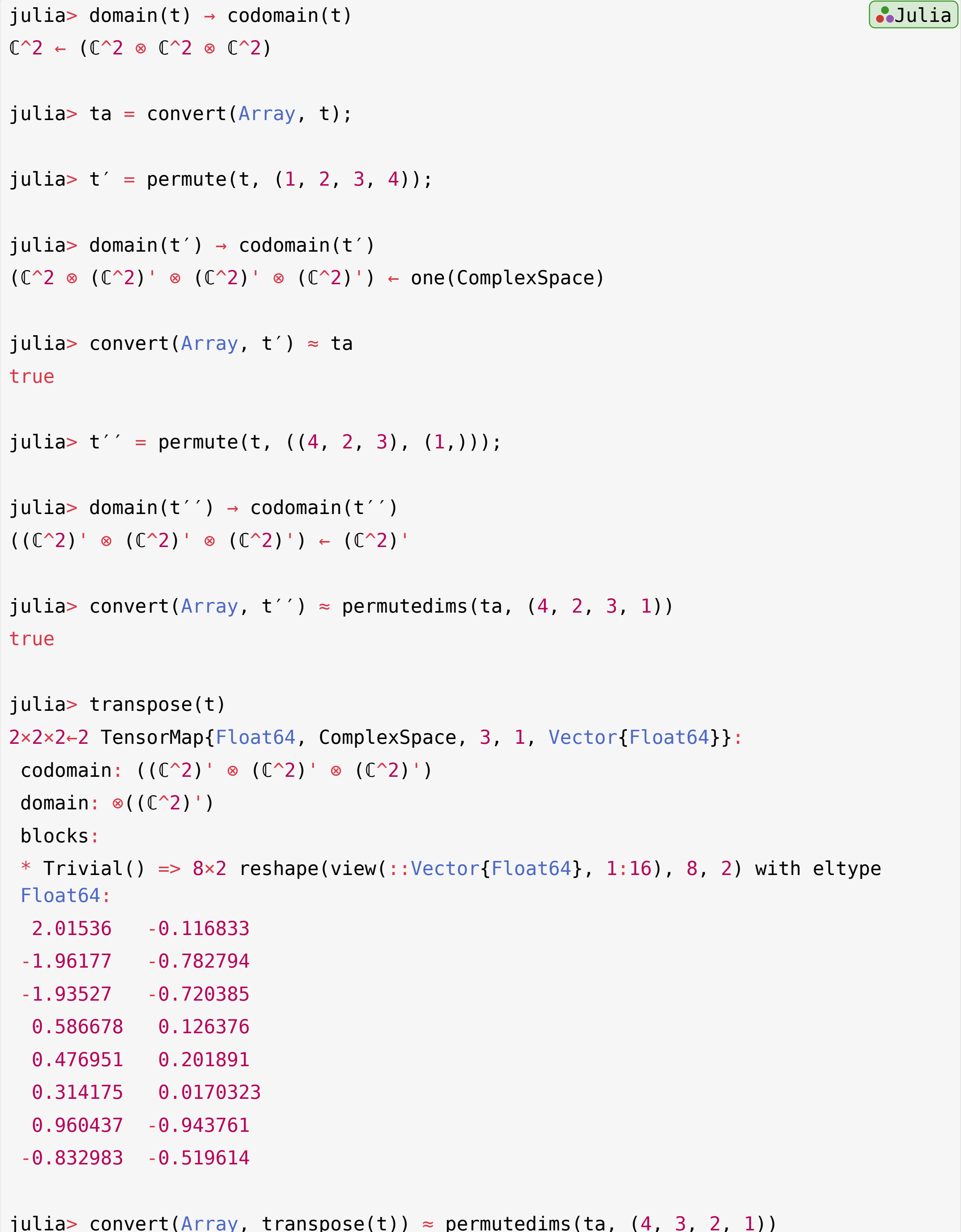
In categorical language, the reason for this extra twist is that we use the left coevaluation η, but the right evaluation $\tilde{\epsilon}$, when repartitioning the indices between domain and codomain.

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 nontrivial on the fusion trees and on the corresponding data, after conversion to a regular `Array` (when possible), it just acts like `permutedims`



```julia
julia> domain(t) → codomain(t)
ℂ^2 ← (ℂ^2 ⊗ ℂ^2 ⊗ ℂ^2)

julia> ta = convert(Array, t);

julia> t′ = permute(t, (1, 2, 3, 4));

julia> domain(t′) → codomain(t′)
(ℂ^2 ⊗ (ℂ^2)' ⊗ (ℂ^2)' ⊗ (ℂ^2)') ← one(ComplexSpace)

julia> convert(Array, t′) ≈ ta
true

julia> t′′ = permute(t, ((4, 2, 3), (1,)));

julia> domain(t′′) → codomain(t′′)
((ℂ^2)' ⊗ (ℂ^2)' ⊗ (ℂ^2)') ← (ℂ^2)'

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

julia> transpose(t)
2×2×2←2 TensorMap{Float64, ComplexSpace, 3, 1, Vector{Float64}}:
 codomain: ((ℂ^2)' ⊗ (ℂ^2)' ⊗ (ℂ^2)')
 domain: ⊗((ℂ^2)')
 blocks:
 * Trivial() => 8×2 reshape(view(::Vector{Float64}, 1:16), 8, 2) with eltype
 Float64:
  2.01536   -0.116833
 -1.96177   -0.782794
 -1.93527   -0.720385
  0.586678   0.126376
  0.476951   0.201891
  0.314175   0.0170323
  0.960437  -0.943761
 -0.832983  -0.519614

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
```

Note that `transpose` acts like one would expect on a `TensorMap{T, S, 1, 1}` . On a `TensorMap{T, 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 multi-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
julia> V1 = GradedSpace{FibonacciAnyon}(:I => 3, :τ => 2)
Vect[FibonacciAnyon](…) of dim 6.23606797749979:
 :I => 3
 :τ => 2

julia> V2 = GradedSpace{FibonacciAnyon}(:I => 2, :τ => 1)
Vect[FibonacciAnyon](…) of dim 3.618033988749895:
 :I => 2
 :τ => 1

julia> m = randn(Float32, V1, V2)
┌ Warning: Tensors with real data might be incompatible with sector type
FibonacciAnyon
└ @ TensorKit ~/Desktop/undergrad/10-26spring/code/TensorKit.jl/src/tensors/
tensor.jl:32
6.23606797749979←3.618033988749895 TensorMap{Float32, Vect[FibonacciAnyon], 1, 1,
Vector{Float32}}:
 codomain: ⊗(Vect[FibonacciAnyon](:I => 3, :τ => 2))
 domain: ⊗(Vect[FibonacciAnyon](:I => 2, :τ => 1))
 blocks:
```

```
 * FibonacciAnyon(:I) => 3×2 reshape(view(::Vector{Float32}, 1:6), 3, 2) with
 eltype Float32:
 -0.503904    0.93728
  0.0204966  -0.246134
 -0.550483   -0.637508

 * FibonacciAnyon(:τ) => 2×1 reshape(view(::Vector{Float32}, 7:8), 2, 1) with
 eltype Float32:
  1.4091314
 -1.9744852

julia> transpose(m)
3.618033988749895←6.23606797749979 TensorMap{ComplexF32, Vect[FibonacciAnyon], 1,
1, Vector{ComplexF32}}:
 codomain: ⊗(Vect[FibonacciAnyon](:I => 2, :τ => 1)')
 domain: ⊗(Vect[FibonacciAnyon](:I => 3, :τ => 2)')
 blocks:
 * FibonacciAnyon(:I) => 2×3 reshape(view(::Vector{ComplexF32}, 1:6), 2, 3) with
 eltype ComplexF32:
 -0.503904+0.0im  0.0204966+0.0im  -0.550483+0.0im
   0.93728+0.0im  -0.246134+0.0im  -0.637508+0.0im

 * FibonacciAnyon(:τ) => 1×2 reshape(view(::Vector{ComplexF32}, 7:8), 1, 2) with
 eltype ComplexF32:
 1.40913+0.0im  -1.97449+0.0im

julia> twist(braid(m, ((2,), (1,)), (1, 2)), 1)
3.618033988749895←6.23606797749979 TensorMap{ComplexF32, Vect[FibonacciAnyon], 1,
1, Vector{ComplexF32}}:
 codomain: ⊗(Vect[FibonacciAnyon](:I => 2, :τ => 1)')
 domain: ⊗(Vect[FibonacciAnyon](:I => 3, :τ => 2)')
 blocks:
 * FibonacciAnyon(:I) => 2×3 reshape(view(::Vector{ComplexF32}, 1:6), 2, 3) with
 eltype ComplexF32:
 -0.503904+0.0im  0.0204966+0.0im  -0.550483+0.0im
   0.93728+0.0im  -0.246134+0.0im  -0.637508+0.0im

 * FibonacciAnyon(:τ) => 1×2 reshape(view(::Vector{ComplexF32}, 7:8), 1, 2) with
 eltype ComplexF32:
 1.40913+1.56148f-8im  -1.97449-3.03648f-8im

julia> t1 = randn(V1 * V2', V2 * V1);
┌ Warning: Tensors with real data might be incompatible with sector type
FibonacciAnyon
└ @ TensorKit ~/Desktop/undergrad/10-26spring/code/TensorKit.jl/src/tensors/
tensor.jl:32
```

```
julia> t2 = 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 `EuclideanProduct` 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.

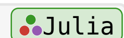
10.3 Tensor factorizations

As tensors are linear maps, they suport various kinds of factorizations. These functions all interpret the provided `AbstractTensorMap` instances as a map from `domain` to `codomain` , which can be thought of as reshaping the tensor into a matrix according to the current bipartition of the indices.

TensorKit's factorizations are provided by MatrixAlgebraKit.jl, which is used to supply both the interface, as well as the implementation of the various operations on the blocks of data. For specific details on the provided functionality, we refer to its documentation page.

Finally, note that each of the factorizations takes the current partition of `domain` and `codomain` as the *axis* along which to matricize and perform the factorization. In order to obtain factorizations according to a different bipartition of the indices, we can use any of the previously mentioned index manipulations before the factorization.

Some examples to conclude this section



```
julia> V1 = SU₂Space(0 => 2, 1/2 => 1)
Rep[SU₂](…) of dim 4:
   0 => 2
 1/2 => 1

julia> V2 = SU₂Space(0 => 1, 1/2 => 1, 1 => 1)
Rep[SU₂](…) of dim 6:
   0 => 1
 1/2 => 1
   1 => 1

julia> t = randn(V1 ⊗ V1, V2);
```

```
julia> U, S, Vh = svd_compact(t);

julia> t ≈ U * S * Vh
true

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

julia> D ≈ S * S
true

julia> U' * U ≈ id(domain(U))
true

julia> S
6←6 DiagonalTensorMap{Float64, Rep[SU₂], Vector{Float64}}:
 codomain: ⊗(Rep[SU₂](0 => 1, 1/2 => 1, 1 => 1))
 domain: ⊗(Rep[SU₂](0 => 1, 1/2 => 1, 1 => 1))
 blocks:
 * Irrep[SU₂](0) => 1×1 LinearAlgebra.Diagonal{Float64, SubArray{Float64, 1,
 Vector{Float64}, Tuple{UnitRange{Int64}}, true}}:
 1.5093667764795946

 * Irrep[SU₂](1/2) => 1×1 LinearAlgebra.Diagonal{Float64, SubArray{Float64, 1,
 Vector{Float64}, Tuple{UnitRange{Int64}}, true}}:
 2.015465414765681

 * Irrep[SU₂](1) => 1×1 LinearAlgebra.Diagonal{Float64, SubArray{Float64, 1,
 Vector{Float64}, Tuple{UnitRange{Int64}}, true}}:
 1.0196988594110132

julia> Q, R = left_orth(t; alg = :svd);

julia> Q' * Q ≈ id(domain(Q))
true

julia> t ≈ Q * R
true

julia> U2, S2, Vh2, ε = svd_trunc(t; trunc = truncspace(V1));

julia> Vh2 * Vh2' ≈ id(codomain(Vh2))
true

julia> S2
```

```
3←3 DiagonalTensorMap{Float64, Rep[SU₂], Vector{Float64}}:
 codomain: ⊗(Rep[SU₂](0 => 1, 1/2 => 1))
 domain: ⊗(Rep[SU₂](0 => 1, 1/2 => 1))
 blocks:
 * Irrep[SU₂](0) => 1×1 LinearAlgebra.Diagonal{Float64, SubArray{Float64, 1,
 Vector{Float64}, Tuple{UnitRange{Int64}}, true}}:
 1.5093667764795946

 * Irrep[SU₂](1/2) => 1×1 LinearAlgebra.Diagonal{Float64, SubArray{Float64, 1,
 Vector{Float64}, Tuple{UnitRange{Int64}}, true}}:
 2.015465414765681

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

julia> L, Q = right_orth(permute(t, ((1,), (2, 3))));

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

julia> Q * Q'
4←4 TensorMap{Float64, Rep[SU₂], 1, 1, Vector{Float64}}:
 codomain: ⊗(Rep[SU₂](0 => 2, 1/2 => 1))
 domain: ⊗(Rep[SU₂](0 => 2, 1/2 => 1))
 blocks:
 * Irrep[SU₂](0) => 2×2 reshape(view(::Vector{Float64}, 1:4), 2, 2) with eltype
 Float64:
  1.0          -1.40725e-16
 -1.40725e-16   1.0

 * Irrep[SU₂](1/2) => 1×1 reshape(view(::Vector{Float64}, 5:5), 1, 1) with eltype
 Float64:
 1.0000000000000002

julia> P = Q' * Q;

julia> P ≈ P * P
true

julia> t′ = permute(t, ((1,), (2, 3)));

julia> t′ ≈ t′ * P
true
```

10.4 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 eminating 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 what is known as *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 TensorKit.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 ligned up horizontally, where the original morphism compositions are now being performed by evaluations ϵ. This process will generate a number of crossings and twists, where the latter 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

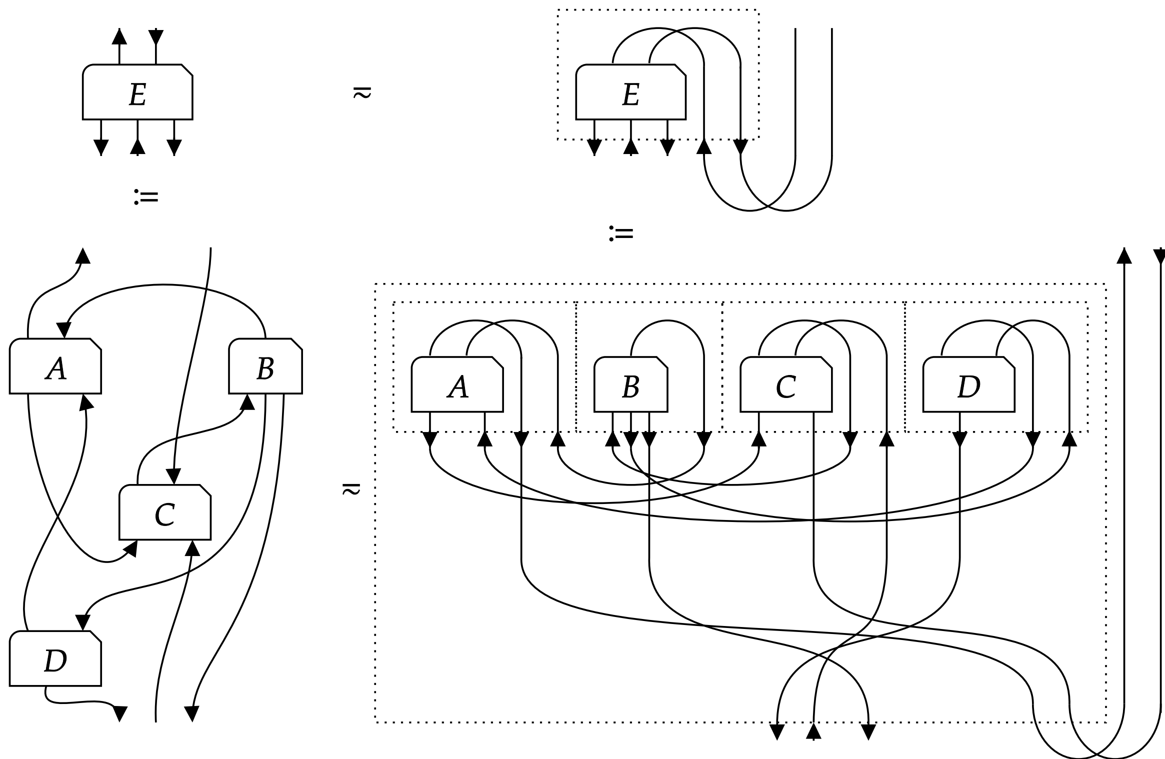
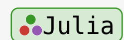


Figure 10.2: tensor unitary

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 convention. 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.

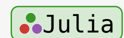
The above picture would be encoded as

```
@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 between `[ ]` is not resolved in the current context but interpreted as a dummy label. Here, we label the indices of a `TensorMap` , like `A::TensorMap{T, 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

```julia
@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

```julia
@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

```julia
@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

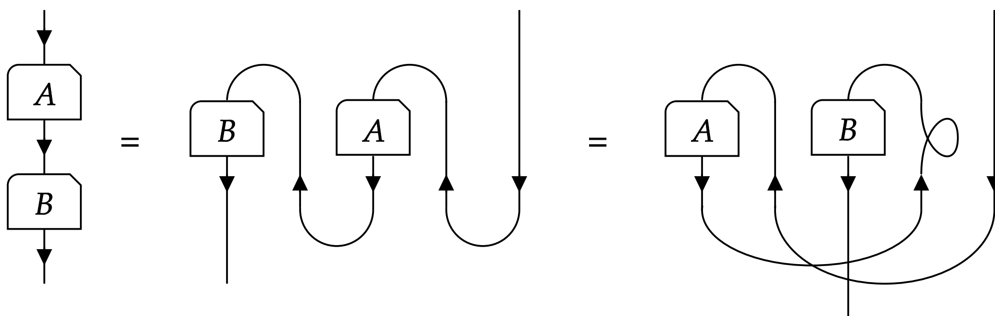


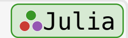
Figure 10.3: tensor contraction reorder

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

```julia
@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 too worried about the actual bipartition into codomain and domain of a given `TensorMap` instance. Indeed, for tensor contractions the `@tensor` macro figures out the correct manipulations automatically. However, when wanting to use the `adjoint` of an instance `t::TensorMap{T, S, N₁, N₂}` , the resulting `adjoint(t)` is an `AbstractTensorMap{T, S, N₂, N₁}` and one needs 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 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` . For example, for `TensorMap{T, S, 1, 1}` instances, this yields exactly the equivalence one expects, namely one between the following two expressions:



```julia
@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{T, 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:

10.5 Fermionic tensor contractions

TODO

10.6 Anyonic tensor contractions

TODO

Part III

Library

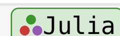
Chapter 11

Symmetry sectors

11.1 Type hierarchy

The fundamental abstract supertype for symmetry sectors is `Sector` :

`TensorKitSectors.Sector` – Type.



```julia
abstract type Sector
```

Abstract type for representing the (isomorphism classes of) simple objects in (unitary and pivotal) (pre-)fusion categories, e.g. the irreducible representations of a finite or compact group. Subtypes `I <: Sector` as the set of labels of a `GradedSpace` .

Every new `I <: Sector` should implement the following methods:

- `unit(::Type{I})` : unit element of `I` . If there are multiple, implement `allunits(::Type{I})` instead.
- `dual(a::I)` : $\bar{a}$, conjugate or dual label of a
- `⊗(a::I, b::I)` : iterable with unique fusion outputs of $a \otimes b$ (i.e. don't repeat in case of multiplicities)
- `Nsymbol(a::I, b::I, c::I)` : number of times `c` appears in `a ⊗ b` , i.e. the multiplicity
- `FusionStyle(::Type{I})` : `UniqueFusion()` , `SimpleFusion()` or `GenericFusion()`
- `BraidingStyle(::Type{I})` : `Bosonic()` , `Fermionic()` , `Anyonic()` , ...
- `Fsymbol(a::I, b::I, c::I, d::I, e::I, f::I)` : F-symbol: scalar (in case of `UniqueFusion` / `SimpleFusion`) or rank-4 array (in case of `GenericFusion`)
- `Rsymbol(a::I, b::I, c::I)` : R-symbol: scalar (in case of `UniqueFusion` / `SimpleFusion`) or matrix (in case of `GenericFusion`)
- `isless(a::I, b::I)` : defines a canonical ordering of sectors
- `hash(a::I)` : hash function for sectors

and optionally

- `dim(a::I)` : quantum dimension of sector `a`
- `frobenius_schur_indicator(a::I)` : Frobenius-Schur indicator of `a` (1, 0, −1)
- `frobenius_schur_phase(a::I)` : Frobenius-Schur phase of `a` (±1)
- `sectorscalartype(::Type{I})` : scalar type of F- and R-symbols
- `Bsymbol(a::I, b::I, c::I)` : B-symbol: scalar (in case of `UniqueFusion` / `SimpleFusion`) or matrix (in case of `GenericFusion`)
- `twist(a::I)` -> twist of sector `a`

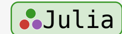
Furthermore, `iterate` and `Base.IteratorSize` should be made to work for the singleton type `SectorValues{I}` .

To help with the implementation of `⊗(a::I, b::I)` as an iterator, the provided `struct` type `SectorProductIterator{I}` can be used, which stores `a` and `b` and requires the implementation of `Base.iterate(::SectorProductIterator{I}, state...)` .

source

Various concrete subtypes of `Sector` are provided within the TensorKitSectors library:

`TensorKitSectors.Trivial` – Type.

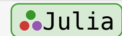


```julia
struct Trivial <: Sector
Trivial()
```

Singleton type to represent the trivial sector, i.e. the trivial representation of the trivial group. This is equivalent to `Rep[ℤ₁]` , or the unit object of the category `Vect` of ordinary vector spaces.

source

`TensorKitSectors.AbstractIrrep` – Type.



```julia
abstract type AbstractIrrep{G <: Group} <: Sector
```

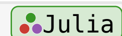
Abstract supertype for sectors which corresponds to irreps (irreducible representations) of a group `G` . As we assume unitary representations, these would be finite groups or compact Lie groups. Note that this could also include projective rather than linear representations.

Actual concrete implementations of those irreps can be obtained as `Irrep[G]` , or via their actual name, which generically takes the form `(asciiG)Irrep` , i.e. the ASCII spelling of the group name followed by `Irrep` .

All irreps have `BraidingStyle` equal to `Bosonic()` and thus trivial twists.

source

`TensorKitSectors.ZNIrrep` – Type.



```julia
struct ZNIrrep{N} <: AbstractIrrep{ℤ{N}}
ZNIrrep{N}(n::Integer)
Irrep[ℤ{N}](n::Integer)
```

Represents irreps of the group $\mathbb{Z}_N$ for some value of `N` . For `N` equals `2` , `3` or `4` , `ℤ{N}` can be replaced by `ℤ₂` , `ℤ₃` , and `ℤ₄` . An arbitrary `Integer` `n` can be provided to the constructor, but only the value `mod(n, N)` is relevant.

The type of the stored integer (`UInt8`) requires `N ≤ 128` . Larger values of `N` should use the `LargeZNIrrep` instead. The constructor `Irrep[ℤ{N}]` should be preferred, as it will automatically select the most efficient storage type for a given value of `N` .

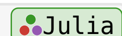
See also `charge` and `modulus` to extract the relevant data.

Fields

- `n::UInt8` : the integer label of the irrep, modulo `N` .

source

`TensorKitSectors.DNIrrep` – Type.



```julia
struct DNIrrep{N} <: AbstractIrrep{Dihedral{N}}
```

```
DNIrrep{N}(n::Integer, isodd::Bool=false)
Irrep[Dihedral{N}](n::Integer, isodd::Bool=false)
```

Represents irreps of the dihedral group $D_N = Z_N \rtimes C$ (Z_N and charge conjugation or reflection).

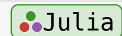
Properties

- `j::Int` : the value of the Z_N charge.
- `isodd::Bool` : the representation of charge conjugation.

Combined these take the values $+0, -0, 1, ..., (N-1)/2$ for odd N, and $+0, -0, 1, ..., N/2 - 1, +(N/2), -(N/2)$ for even N, where the $+$ ($-$) refer to the even (odd) one-dimensional irreps, while the others are two-dimensional.

source

`TensorKitSectors.U1Irrep` – Type.



```
struct U1Irrep <: AbstractIrrep{U₁}
U1Irrep(charge::Real)
Irrep[U₁](charge::Real)
```

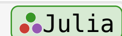
Represents irreps of the group U_1. The irrep is labelled by a charge, which should be an integer for a linear representation. However, it is often useful to allow half integers to represent irreps of U_1 subgroups of SU_2, such as the S^z of spin-1/2 system. Hence, the charge is stored as a `HalfInt` from the package HalfIntegers.jl, but can be entered as arbitrary `Real` . The sequence of the charges is: 0, 1/2, −1/2, 1, −1, …

Fields

- `charge::HalfInt` : the label of the irrep, which can be any half integer.

source

`TensorKitSectors.SU2Irrep` – Type.



```
struct SU2Irrep <: AbstractIrrep{SU₂}
SU2Irrep(j::Real)
Irrep[SU₂](j::Real)
```

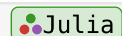
Represents irreps of the group SU_2. The irrep is labelled by a half integer `j` which can be entered as an abitrary `Real` , but is stored as a `HalfInt` from the HalfIntegers.jl package.

Fields

- `j::HalfInt` : the label of the irrep, which can be any non-negative half integer.

source

`TensorKitSectors.CU1Irrep` – Type.



```
struct CU1Irrep <: AbstractIrrep{CU₁}
CU1Irrep(j, s = ifelse(j>zero(j), 2, 0))
Irrep[CU₁](j, s = ifelse(j>zero(j), 2, 0))
```

Represents irreps of the group $U_1 \rtimes C$ (U_1 and charge conjugation or reflection), which is also known as just `O₂` .

Fields

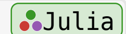
- `j::HalfInt` : the value of the U_1 charge.
- `s::Int` : the representation of charge conjugation.

They can take values:

- if `j == 0` , `s = 0` (trivial charge conjugation) or `s = 1` (non-trivial charge conjugation)
- if `j > 0` , `s = 2` (two-dimensional representation)

source

`TensorKitSectors.AbstractGroupElement` – Type.

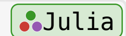
```julia
abstract type AbstractGroupElement{G <: Group} <: Sector
```


Abstract supertype for sectors which corresponds to group elements of a group `G` .

Actual concrete implementations of those irreps can be obtained as `Element[G]` , or via their actual name, which generically takes the form `(asciiG)Element` , i.e. the ASCII spelling of the group name followed by `Element` .

All group elements have `FusionStyle` equal to `UniqueFusion()` . Furthermore, the `BraidingStyle` is set to `NoBraiding()` , although this can be overridden by a concrete implementation of `AbstractGroupElement` .

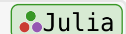
For the fusion structure, a specific `SomeGroupElement <: AbstractGroupElement{SomeGroup}` should only implement the following methods

```julia
Base.:*(c1::GroupElement, c2::GroupElement) -> GroupElement
Base.one(::Type{GroupElement}) -> GroupElement
Base.inv(c::GroupElement) -> GroupElement
# and optionally
TensorKitSectors.cocycle(c1::GroupElement, c2::GroupElement, c3::GroupElement)
-> Number
```


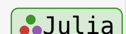
The methods `conj` , `dual` , `⊗` , `Nsymbol` , `Fsymbol` , `dim` , `Asymbol` , `Bsymbol` and `frobenius_schur_phase` will then be automatically defined. If no `cocycle` method is defined, the cocycle will be assumed to be trivial, i.e. equal to `1` .

source

`TensorKitSectors.ZNElement` – Type.

```julia
struct ZNElement{N, p} <: AbstractGroupElement{ℤ{N}}
ZNElement{N, p}(n::Integer)
GroupElement[ℤ{N}, p](n::Integer)
```


Represents an element of the group $\mathbb{Z}_N$ for some value of `N < 64` . (We need `2 * (N - 1) <= 127` in order for `a ⊗ b` to work correctly.) For `N` equals `2` , `3` or `4` , `ℤ{N}` can be replaced by `ℤ₂` , `ℤ₃` , `ℤ₄` . An arbitrary `Integer` `n` can be provided to the constructor, but only the value `mod(n, N)` is relevant. The second type parameter `p` should also be specified as an integer `0 <= p < N` and specifies the 3-cocycle, which is then being given by

```julia
cocycle(a, b, c) = cispi(2 * p * a.n * (b.n + c.n - mod(b.n + c.n,
N)) / N))
```


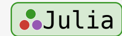
If `p` is not specified, it defaults to `0` , i.e. the trivial cocycle.

Fields

- `n::Int8` : the integer label of the element, modulo `N` .

source

`TensorKitSectors.FermionParity` – Type.



```
struct FermionParity <: Sector
FermionParity(isodd::Bool)
```

Represents sectors with fermion parity. The fermion parity is a $\mathbb{Z}_2$ quantum number that yields an additional sign when two odd fermions are exchanged, corresponding to a `BraidingStyle` that is `Fermionic()` .

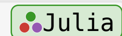
Fields

- `isodd::Bool` : indicates whether the fermion parity is odd (`true`) or even (`false`).

See also: `FermionNumber` , `FermionSpin`

source

`TensorKitSectors.FermionNumber` – Type.



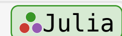
```
const FermionNumber = U1Irrep ⊠ FermionParity
FermionNumber(a::Int)
```

Represents the fermion number as the direct product of a U_1 irrep `a` and a fermion parity, with the restriction that the fermion parity is odd if and only if `a` is odd.

See also: `U1Irrep` , `FermionParity`

source

`TensorKitSectors.FermionSpin` – Type.



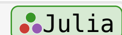
```
const FermionSpin = SU2Irrep ⊠ FermionParity
FermionSpin(j::Real)
```

Represents the fermion spin as the direct product of a SU_2 irrep `j` and a fermion parity, with the restriction that the fermion parity is odd if `2 * j` is odd.

See also: `SU2Irrep` , `FermionParity`

source

`TensorKitSectors.FibonacciAnyon` – Type.



```
struct FibonacciAnyon <: Sector
FibonacciAnyon(s::Symbol)
```

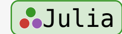
Represents the anyons of the Fibonacci modular fusion category. It can take two values, corresponding to the trivial sector `FibonacciAnyon(:I)` and the non-trivial sector `FibonacciAnyon(:τ)` with fusion rules $\tau \otimes \tau = 1 \oplus \tau$.

Fields

- `isunit::Bool` : indicates whether the sector corresponds to the trivial anyon `:I` (`true`), or the non-trivial anyon `:τ` (`false`).

source

`TensorKitSectors.IsingAnyon` – Type.



```
struct IsingAnyon <: Sector
IsingAnyon(s::Symbol)
```

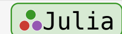
Represents the anyons of the Ising modular fusion category. It can take three values, corresponding to the trivial sector `IsingAnyon(:I)` and the non-trivial sectors `IsingAnyon(:σ)` and `IsingAnyon(:ψ)` , with fusion rules $\psi \otimes \psi = 1$, $\sigma \otimes \psi = \sigma$, and $\sigma \otimes \sigma = 1 \oplus \psi$.

Fields

- `s::Symbol` : the label of the represented anyon, which can be `:I` , `:σ` , or `:ψ` .

source

`TensorKitSectors.PlanarTrivial` – Type.

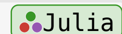


```
struct PlanarTrivial <: Sector
PlanarTrivial()
```

Represents a trivial anyon sector, i.e. a trivial sector without braiding. This is mostly useful for testing.

source

`TensorKitSectors.IsingBimodule` – Type.

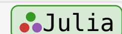


```
struct IsingBimodule <: Sector
```

Type to represent the simple objects in the Ising category reinterpreted as a bimodule category composed of two copies of the category ? = ? = Irrep[$\mathbb{Z}_2$], the two simple objects of which can be identified with the Ising anyons {I, ψ}, and the bimodule categories ? = $?^{\mathrm{op}}$ = Vec, with a single simple object that can be identified with the Ising anyon σ. This constitutes the easiest example of a multifusion category and is implemented here for testing purposes and to illustrate how to implement such categories in TensorKitSectors.jl.

source

`TensorKitSectors.TimeReversed` – Type.

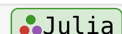


```
struct TimeReversed{I <: Sector}
TimeReversed(a::Sector)
timereversed(a::Sector)
```

Represents the time-reversed version of the sector `a` , i.e. the sector with the same fusion rules and `F` -symbols, but with the inverse braiding. Time reversal acts trivially on sectors with symmetric braiding. In such cases, `timereversed(a)` simply returns `a` .

source

`TensorKitSectors.ProductSector` – Type.



```
struct ProductSector{T <: SectorTuple}
```

```
ProductSector((s₁, s₂, ...))
```

Represents the Deligne tensor product of sectors. The type parameter `T` is a tuple of the component sectors. The recommended way to construct a `ProductSector` is using the `deligneproduct` (`⊠`) operator on the components.

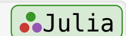
source

Several more concrete sector types can be found in other packages such as SUNRepresentations.jl, CategoryData.jl, QWignerSymbols.jl, ...:

Some of these types are parameterized by a type parameter that represents a group. We therefore also provide a number of types to represent groups:

`TensorKitSectors.Group` – Type.

```
abstract type Group
```

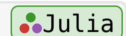


Abstract supertype for representing different types of groups. Groups can be used to define Sector subtypes, either via their irreducible representations, or via their group elements, and typically appear as type parameter. As such, they are not meant to be instantiated and are defined as abstract types.

source

`TensorKitSectors.AbelianGroup` – Type.

```
abstract type AbelianGroup <: Group
```

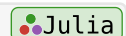


Abstract supertype for representing different types of Abelian groups. Abelian groups have both irreps and group elements that have several simplified properties, that can be defined in general.

source

`TensorKitSectors.Cyclic` – Type.

```
abstract type Cyclic{N} <: AbelianGroup
```

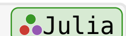


Type to represent the cyclic group of order `N` , i.e. the multiplicative group of roots of unity of order `N` , which is a discrete abelian group. The cyclic group of order `N` is isomorphic to the additive group `ℤ{N}` , and we define the latter as a type alias `const ℤ{N} = Cyclic{N}` .

source

`TensorKitSectors.U₁` – Type.

```
abstract type U₁ <: AbelianGroup
```

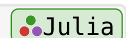


Type to represent the group $U(1)$ of complex numbers of unit modulus, which is a compact Abelian Lie group.

source

`TensorKitSectors.CU₁` – Type.

```
abstract type CU₁ <: Group
```



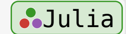
Type to represent the group of U_1 in combination with charge conjugation, i.e. the group generated by U_1 and an additional element that acts as complex conjugation on U_1. This group is

isomorphic to the orthogonal group O_2 of real orthogonal 2×2 matrices, and can be seen as the semidirect product U_1 ? $\mathbb{Z}_2$. This is a compact non-Abelian group.

source

`TensorKitSectors.SU` – Type.

```julia
abstract type SU{N} <: Group
```

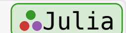


Type to represent the special unitary group $SU(N)$, which is a compact non-Abelian Lie group.

source

`TensorKitSectors.Dihedral` – Type.

```julia
abstract type Dihedral{N} <: Group
```

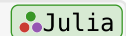


Type to represent the dihedral group of order `2N` , which is the symmetry group of a regular polygon with `N` sides, and is a discrete non-Abelian group.

source

`TensorKitSectors.ProductGroup` – Type.

```julia
abstract type ProductGroup{T <: Tuple{Vararg{Group}}} <: Group
```



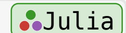
Type to represent the direct product of a tuple of groups. This is typically constructed via the × operator.

source

The following types are used to characterize different properties of the different types of sectors:

`TensorKitSectors.FusionStyle` – Type.

```julia
abstract type FusionStyle
FusionStyle(::Sector)
FusionStyle(I::Type{<:Sector})
```



Trait to describe the fusion behavior of sectors of type `I` , which can be either

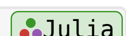
- `UniqueFusion()` : single fusion output when fusing two sectors;
- `SimpleFusion()` : multiple outputs, but every output occurs at most one, also known as multiplicity-free (e.g. irreps of $SU(2)$);
- `GenericFusion()` : multiple outputs that can occur more than once (e.g. irreps of $SU(3)$).

There is an abstract supertype `MultipleFusion` of which both `SimpleFusion` and `GenericFusion` are subtypes. Furthermore, there is a type alias `MultiplicityFreeFusion` for those fusion types which do not require muliplicity labels, i.e. `MultiplicityFreeFusion = Union{UniqueFusion,SimpleFusion}` .

source

`TensorKitSectors.BraidingStyle` – Type.

```julia
abstract type BradingStyle
BraidingStyle(::Sector) -> ::BraidingStyle
BraidingStyle(I::Type{<:Sector}) -> ::BraidingStyle
```



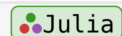
Return the type of braiding and twist behavior of sectors of type `I` , which can be either

- `NoBraiding()` : no braiding structure
- `Bosonic()` : symmetric braiding with trivial twist (i.e. identity)
- `Fermionic()` : symmetric braiding with non-trivial twist (squares to identity)
- `Anyonic()` : general R^{ab}_c phase or matrix (depending on `SimpleFusion` or `GenericFusion` fusion) and arbitrary twists

Note that `Bosonic` and `Fermionic` are subtypes of `SymmetricBraiding` , which means that braids are in fact equivalent to crossings (i.e. braiding twice is an identity: `isone(Rsymbol(b,a,c)*Rsymbol(a,b,c)) == true`) and permutations are uniquely defined.

source

`TensorKitSectors.UnitStyle` – Type.



```
abstract type UnitStyle
UnitStyle(::Sector)
UnitStyle(I::Type{<:Sector})
```

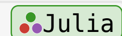
Trait to describe the semisimplicity of the unit sector of type `I` . This can be either

- `SimpleUnit()` : the unit is simple (e.g. fusion categories);
- `GenericUnit()` : the unit is semisimple.

source

Finally, the following auxiliary types are defined to facilitate the implementation of some of the methods on sectors:

`TensorKitSectors.SectorValues` – Type.



```
struct SectorValues{I <: Sector}
```

Singleton type to represent an iterator over the possible values of type `I` , whose instance is obtained as `values(I)` . For a new `I::Sector` , the following should be defined

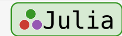
- `Base.iterate(::SectorValues{I}[, state])` : iterate over the values
- `Base.IteratorSize(::Type{SectorValues{I}})` : `HasLength()` , `SizeUnknown()` or `IsInfinite()` depending on whether the number of values of type `I` is finite (and sufficiently small) or infinite; for a large number of values, `SizeUnknown()` is recommended because this will trigger the use of `GenericGradedSpace` .

If `IteratorSize(I) == HasLength()` , also the following must be implemented:

- `Base.length(::SectorValues{I})` : the number of different values
- `Base.getindex(::SectorValues{I}, i::Int)` : a mapping between an index `i` and an instance of `I` . A fallback implementation exists that returns the `i` th value of the `SectorValues` iterator.
- `findindex(::SectorValues{I}, c::I)` : reverse mapping between a value `c::I` and an index `i::Integer ∈ 1:length(values(I))` . A fallback implementation exists that linearly searches through the `SectorValues` iterator.

source

`TensorKitSectors.SectorProductIterator` – Type.



```julia
struct SectorProductIterator{I <: Sector}
SectorProductIterator(a::I, b::I) where {I <: Sector}
```

Custom iterator to represent the (unique) fusion outputs of $a \otimes b$.

Custom sectors that aim to use this have to provide the following functionality:

- `Base.iterate(::SectorProductIterator{I}, state...) where {I <: Sector}` : iterate over the fusion outputs of `a ⊗ b`

If desired and it is possible to easily compute the number of unique fusion outputs, it is also possible to define `Base.IteratorSize(::Type{SectorProductIterator{I}}) = Base.HasLength()` , in which case `Base.length(::SectorProductIterator{I})` has to be implemented.

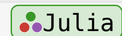
See also `⊗` .

source

11.2 Useful constants

The following constants are defined to facilitate obtaining the type associated with the group elements or the irreducible representations of a given group:

`TensorKitSectors.Irrep` – Constant.

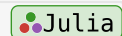


```julia
const Irrep
```

A constant of a singleton type used as `Irrep[G]` with `G <: Group` a type of group, to construct or obtain a concrete subtype of `AbstractIrrep{G}` that implements the data structure used to represent irreducible representations of the group `G` .

source

`TensorKitSectors.GroupElement` – Constant.



```julia
const GroupElement
```

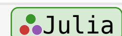
A constant of a singleton type used as `GroupElement[G]` or `GroupElement[G, ω]` with `G <: Group` a type of group, to construct or obtain a concrete subtype of `AbstractElement{G}` that implements the data structure used to represent elements of the group `G` , possibly with a second argument `ω` that specifies the associated 3-cocycle.

source

11.3 Methods for characterizing and manipulating `Sector` objects

The following methods can be used to obtain properties such as topological data of sector objects, or to manipulate them or create related sectors:

`TensorKitSectors.unit` – Function.

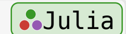


```julia
unit(::Sector) -> Sector
unit(::Type{<:Sector}) -> Sector
```

Return the unit element of this type of sector, provided it is unique.

source

`TensorKitSectors.isunit` – Function.

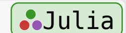


```
isunit(a::Sector) -> Bool
```

Return whether sector `a` is a unit element.

source

`TensorKitSectors.leftunit` – Function.

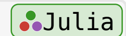


```
leftunit(a::Sector) -> Sector
```

Return the left unit element corresponding to `a` ; this is necessary for multifusion categories, where the unit may not be unique. See also `rightunit` and `unit` .

source

`TensorKitSectors.rightunit` – Function.

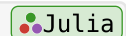


```
rightunit(a::Sector) -> Sector
```

Return the right unit element corresponding to `a` ; this is necessary for multifusion categories, where the unit may not be unique. See also `leftunit` and `unit` .

source

`TensorKitSectors.allunits` – Function.

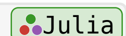


```
allunits(I::Type{<:Sector}) -> Tuple{I}
```

Return a tuple with all units of the sector type `I` . For fusion categories, this will contain only one element.

source

`TensorKitSectors.dual` – Method.

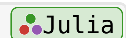


```
dual(a::Sector) -> Sector
```

Return the dual label of `a` , i.e. the unique label `ā = dual(a)` such that `Nsymbol(a, ā, leftunit(a)) == 1` and `Nsymbol(ā, a, rightunit(a)) == 1` .

source

`TensorKitSectors.Nsymbol` – Function.

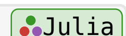


```
Nsymbol(a::I, b::I, c::I) where {I <: Sector} -> Integer
```

Return an `Integer` representing the number of times `c` appears in the fusion product `a ⊗ b` . Could be a `Bool` if `FusionStyle(I) == UniqueFusion()` or `SimpleFusion()` .

source

`TensorKitSectors.:⊗` – Function.



```
⊗(a::I, b::I...) where {I <: Sector}
otimes(a::I, b::I...) where {I <: Sector}
```

Return an iterable of elements of `c::I` that appear in the fusion product `a ⊗ b` .

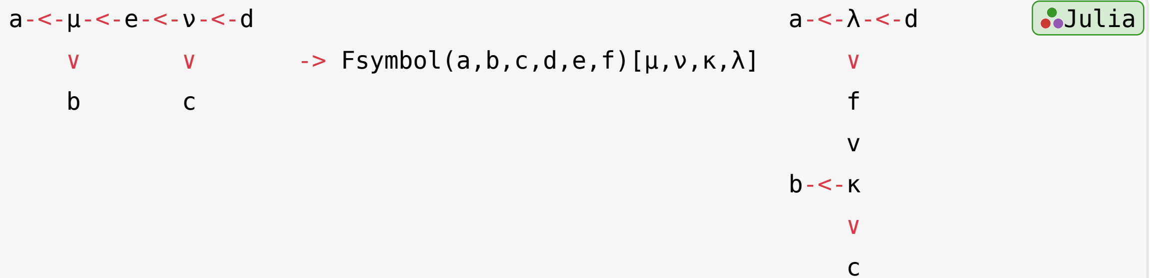
Note that every element `c` should appear at most once, fusion degeneracies (if `FusionStyle(I) == GenericFusion()`) should be accessed via `Nsymbol(a, b, c)` .

source

`TensorKitSectors.Fsymbol` – Function.

```julia
Fsymbol(a::I, b::I, c::I, d::I, e::I, f::I) where {I <: Sector}
```

Return the F-symbol F^{abc}_d that associates the two different fusion orders of sectors `a` , `b` and `c` into an ouput sector `d` , using either an intermediate sector $a \otimes b \to e$ or $b \otimes c \to f$:

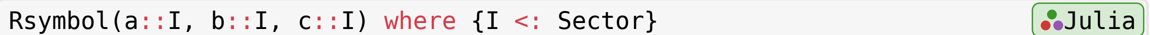


```julia
a-<-μ-<-e-<-ν-<-d                                     a-<-λ-<-d
    ∨       ∨       -> Fsymbol(a,b,c,d,e,f)[μ,ν,κ,λ]      ∨
    b       c                                             f
                                                          v
                                                      b-<-κ
                                                          ∨
                                                          c
```

If `FusionStyle(I)` is `UniqueFusion` or `SimpleFusion` , the F-symbol is a number. Otherwise it is a rank 4 array of size `(Nsymbol(a, b, e), Nsymbol(e, c, d), Nsymbol(b, c, f), Nsymbol(a, f, d))` .

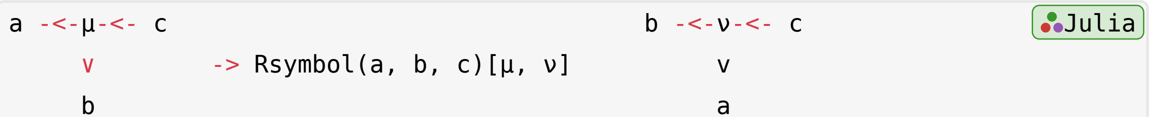
source

`TensorKitSectors.Rsymbol` – Function.



```julia
Rsymbol(a::I, b::I, c::I) where {I <: Sector}
```

Returns the R-symbol R^{ab}_c that maps between $c \to a \otimes b$ and $c \to b \otimes a$ as in

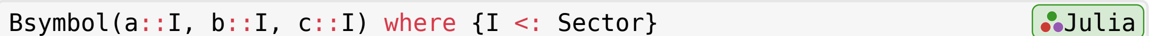


```julia
a -<-μ-<- c                                 b -<-ν-<- c
     ∨         -> Rsymbol(a, b, c)[μ, ν]         v
     b                                           a
```

If `FusionStyle(I)` is `UniqueFusion()` or `SimpleFusion()` , the R-symbol is a number. Otherwise it is a square matrix with row and column size `Nsymbol(a, b, c) == Nsymbol(b, a, c)` .

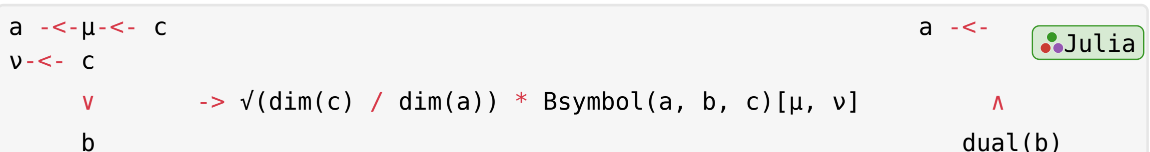
source

`TensorKitSectors.Bsymbol` – Function.



```julia
Bsymbol(a::I, b::I, c::I) where {I <: Sector}
```

Return the value of B^{ab}_c which appears in transforming a splitting vertex into a fusion vertex using the transformation

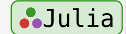


```julia
a -<-μ-<- c                                                  a -<-
ν-<- c
     ∨        -> √(dim(c) / dim(a)) * Bsymbol(a, b, c)[μ, ν]      ∧
     b                                                         dual(b)
```

If `FusionStyle(I)` is `UniqueFusion()` or `SimpleFusion()` , the B-symbol is a number. Otherwise it is a square matrix with row and column size `Nsymbol(a, b, c) == Nsymbol(c, dual(b), a)` .

source

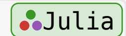
`TensorKitSectors.dim` – Method.

```
dim(a::Sector)
```


Return the (quantum) dimension of the sector `a` .

source

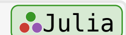
`TensorKitSectors.frobenius_schur_phase` – Function.

```
frobenius_schur_phase(a::Sector)
```


Return the Frobenius-Schur phase κ_a of a sector a, which is a complex phase that appears in the context of bending lines and is obtained from $F^{a\bar{a}a}_a$. When `a == dual(a)` , it is restricted to $\kappa_a \in \{1, -1\}$ and coincides with the group-theoretic version `frobenius_schur_indicator` . When `a != dual(a)` , the value of κ_a can be gauged to be `1` , though is not required to be.

source

`TensorKitSectors.frobenius_schur_indicator` – Function.

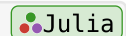
```
frobenius_schur_indicator(a::Sector)
```


Return the Frobenius-Schur indicator of a sector $\nu_a \in \{1, 0, -1\}$, which distinguishes between real, complex and quaternionic representations.

See also `frobenius_schur_phase` for the category-theoretic version that appears in the context of line bending.

source

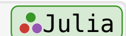
`TensorKitSectors.twist` – Method.

```
twist(a::Sector)
```


Return the twist of a sector `a` .

source

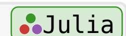
`Base.isreal` – Method.

```
isreal(::Type{<:Sector}) -> Bool
```


Return whether the topological data (Fsymbol, Rsymbol) of the sector is real or not (in which case it is complex).

source

`TensorKitSectors.sectorscalartype` – Function.

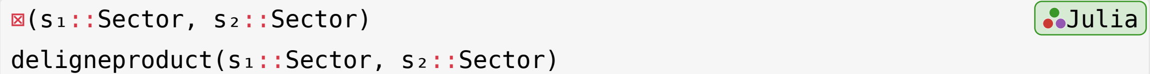
```
sectorscalartype(I::Type{<:Sector}) -> Type{<:Number}
```


Return the scalar type of the topological data of the sector `I` . In particular, this is a combination of the scalar type of both the `Fsymbol` and `Rsymbol` , and determines the scalar type of the `fusiontensor` whenever it is defined.

See also `fusionscalartype` and `braidingscalartype` .

source

`TensorKitSectors.deligneproduct` – Method.



```julia
⊠(s₁::Sector, s₂::Sector)
deligneproduct(s₁::Sector, s₂::Sector)
```

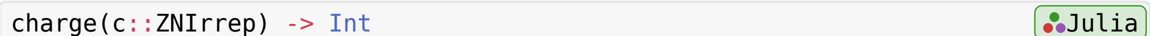
Given two sectors `s₁` and `s₂` , which label an isomorphism class of simple objects in a fusion category C_1 and C_2, `s1 ⊠ s2` (obtained as `\boxtimes+TAB`) labels the isomorphism class of simple objects in the Deligne tensor product category $C_1 \boxtimes C_2$.

The Deligne tensor product also works in the type domain and for spaces and tensors. For group representations, we have `Irrep[G₁] ⊠ Irrep[G₂] == Irrep[G₁ × G₂]` .

source

We have also the following methods that are specific to certain types of sectors and serve as accessors to their fields:

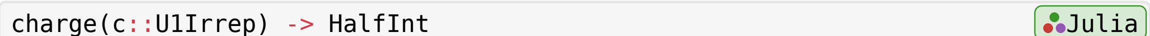
`TensorKitSectors.charge` – Function.



```julia
charge(c::ZNIrrep) -> Int
```

The charge label of the irrep `c` .

source

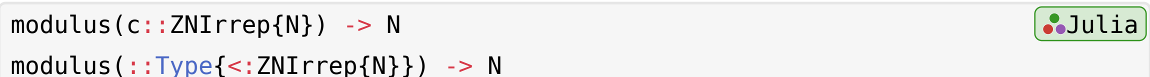


```julia
charge(c::U1Irrep) -> HalfInt
```

The charge label of the irrep `c` .

source

`TensorKitSectors.modulus` – Function.



```julia
modulus(c::ZNIrrep{N}) -> N
modulus(::Type{<:ZNIrrep{N}}) -> N
```

The order of the cyclic group, or the modulus of the charge labels.

source

Furthermore, we also have one specific method acting on groups, represented as types

`TensorKitSectors.:×` – Function.

```julia
×(G::Vararg{Type{<:Group}}) -> ProductGroup{Tuple{G...}}
times(G::Vararg{Type{<:Group}}) -> ProductGroup{Tuple{G...}}
```

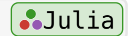
Construct the direct product of a (list of) groups.

source

Mapping between sectors and linear indices is only used for sectors `I` for which `Base.IteratorSize(values(I)) == HasLength()` . In that case, we map an index `i` to a sector `c` via `c = getindex(values(I), i)` , and provide an inverse mapping

`TensorKitSectors.findindex` – Function.

```
findindex(v::SectorValues{I}, c::I)
```



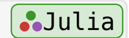
Reverse mapping between a value `c::I` and an index `i::Integer ∈ 1:length(values(I))` .

source

Because we sometimes want to customize the string representation of our sector types, we also have the following method:

`TensorKitSectors.type_repr` – Function.

```
type_repr(T::Type)
```



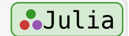
Return a string representation of the type `T` , which is used to modify the default way in which `Sector` subtypes are displayed in other objects that depend on them.

source

Finally, we provide functionality to compile all revelant methods for a sector:

`TensorKitSectors.precompile_sector` – Function.

```
precompile_sector(I::Type{<:Sector})
```



Precompile common methods for the given sector type.

source

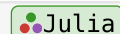
Chapter 12

Fusion trees

Chapter 13

Type hierarchy

`TensorKit.FusionTree` – Type.



```
struct FusionTree{I, N, M, L}
```

Represents a fusion tree of sectors of type `I<:Sector`, fusing (or splitting) `N` uncoupled sectors to a coupled sector. It actually represents a splitting tree, but fusion tree is a more common term.

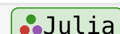
Fields

- `uncoupled::NTuple{N,I}` : the uncoupled sectors coming out of the splitting tree, before the possible ? isomorphism (see `isdual`).
- `coupled::I` : the coupled sector.
- `isdual::NTuple{N,Bool}` : indicates whether a ? isomorphism is present (`true`) or not (`false`) for each uncoupled sector.
- `innerlines::NTuple{M,I}` : the labels of the `M=max(0, N-2)` inner lines of the splitting tree.
- `vertices::NTuple{L,Int}` : the integer values of the `L=max(0, N-1)` vertices of the splitting tree. If `FusionStyle(I) isa MultiplicityFreeFusion`, then `vertices` is simply equal to the constant value `ntuple(n->1, L)`.

source

13.1 Methods for defining and generating fusion trees

`TensorKit.fusiontrees` – Method.



```
fusiontrees(uncoupled::NTuple{N,I}[,
    coupled::I=unit(I)[, isdual::NTuple{N,Bool}=ntuple(n -> false,
    length(uncoupled))]])
    where {N,I<:Sector} -> FusionTreeIterator{I,N,I}
```

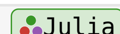
Return an iterator over all fusion trees with a given coupled sector label `coupled` and uncoupled sector labels and isomorphisms `uncoupled` and `isdual` respectively.

source

13.2 Methods for manipulating fusion trees

For manipulating single fusion trees, the following internal methods are defined:

`TensorKit.insertat` – Function.



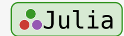
```
insertat(f::FusionTree{I, N₁}, i::Int, f₂::FusionTree{I, N₂})
-> <:AbstractDict{<:FusionTree{I, N₁+N₂-1}, <:Number}
```

Attach a fusion tree `f₂` to the uncoupled leg `i` of the fusion tree `f₁` and bring it into a linear combination of fusion trees in standard form. This requires that `f₂.coupled == f₁.uncoupled[i]` and `f₁.isdual[i] == false`.

source

`TensorKit.split` – Function.

```julia
split(f::FusionTree{I, N}, M::Int) -> (::FusionTree{I,
M}, ::FusionTree{I, N - M + 1})
```



Split a fusion tree into two. The first tree has as uncoupled sectors the first `M` uncoupled sectors of the input tree `f`, whereas its coupled sector corresponds to the internal sector between uncoupled sectors `M` and `M+1` of the original tree `f`. The second tree has as first uncoupled sector that same internal sector of `f`, followed by remaining `N-M` uncoupled sectors of `f`. It couples to the same sector as `f`. This operation is the inverse of `join` in the sense that if `f == join(split(f, M)...)` holds for all `M` between `0` and `N`, where `N` is the number of uncoupled sectors of `f`.

See also `join` and `insertat`.

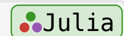
Examples

```julia
julia> f = FusionTree{Z2Irrep}((1, 1, 0), 0, (false, false, false));

julia> f₁, f₂ = TensorKit.split(f, 2)
(FusionTree{Irrep[ℤ₂]}((1, 1), 0, (false, false), ()), FusionTree{Irrep[ℤ₂]}
((0, 0), 0, (false, false), ()))

julia> TensorKit.join(f₁, f₂) == f
true
```

source

`TensorKit.join` – Function.

```julia
join(f₁::FusionTree{I, N₁}, f₂::FusionTree{I, N₂}) where {I, N₁, N₂}
-> (::FusionTree{I, N₁ + N₂ - 1})
```

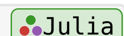


Join fusion trees `f₁` and `f₂` by connecting the coupled sector of `f₁` to the first uncoupled sector of `f₂`. The resulting tree has uncoupled sectors given by those of `f₁` followed the remaining uncoupled sectors (except for the first) of `f₂`. This requires that `f₁.coupled == f₂.uncoupled[1]` and `f₂.isdual[1] == false`. This operation is the inverse of split, in the sense that `f == join(split(f, M)...)` holds for all `M` between `0` and `N`, where `N` is the number of uncoupled sectors of `f`.

See also `split` and `insertat`.

Examples

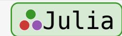
```julia
julia> f₁ = FusionTree{Z2Irrep}((1, 1), 0, (false, false));

julia> f₂ = FusionTree{Z2Irrep}((0, 0), 0, (false, false));

julia> f = TensorKit.join(f₁, f₂)
```



```
FusionTree{Irrep[ℤ₂]}((1, 1, 0), 0, (false, false, false), (0,))
```

source

`TensorKit.merge` – Function.

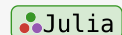


```
merge(f₁::FusionTree{I, N₁}, f₂::FusionTree{I, N₂}, c::I, μ = 1)
-> <:AbstractDict{<:FusionTree{I, N₁+N₂}, <:Number}
```

Merge two fusion trees together to a linear combination of fusion trees whose uncoupled sectors are those of `f₁` followed by those of `f₂` , and where the two coupled sectors of `f₁` and `f₂` are further fused to `c` . In case of `FusionStyle(I) == GenericFusion()` , also a degeneracy label μ for the fusion of the coupled sectors of `f₁` and `f₂` to `c` needs to be specified.

source

`TensorKit.elementary_trace` – Function.

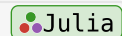


```
elementary_trace(f::FusionTree{I, N}, i) where {I,N} ->
<:AbstractDict{FusionTree{I,N-2}, <:Number}
```

Perform an elementary trace of neighbouring uncoupled indices `i` and `i+1` on a fusion tree `f` , and returns the result as a dictionary of output trees and corresponding coefficients.

source

`TensorKit.planar_trace` – Method.

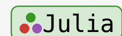


```
planar_trace(f::FusionTree, (q₁, q₂)::Index2Tuple)
    -> <:AbstractDict{<:FusionTree, <:Number}
```

Perform a planar trace of the uncoupled indices of the fusion tree `f` at `q₁` with those at `q₂` , where `q₁[i]` is connected to `q₂[i]` for all `i` . The result is returned as a dictionary of output trees and corresponding coefficients.

source

`TensorKit.artin_braid` – Function.



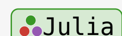
```
artin_braid(f::FusionTree, i; inv::Bool = false) ->
<:AbstractDict{typeof(f), <:Number}
```

Perform an elementary braid (Artin generator) of neighbouring uncoupled indices `i` and `i+1` on a fusion tree `f` , and returns the result as a dictionary of output trees and corresponding coefficients.

The keyword `inv` determines whether index `i` will braid above or below index `i+1` , i.e. applying `artin_braid(f′, i; inv = true)` to all the outputs `f′` of `artin_braid(f, i; inv = false)` and collecting the results should yield a single fusion tree with non-zero coefficient, namely `f` with coefficient `1` . This keyword has no effect if `BraidingStyle(sectortype(f)) isa SymmetricBraiding` .

source

`TensorKit.braid` – Method.



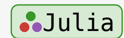
```
braid(f::FusionTree{<:Sector, N}, p::NTuple{N, Int},
levels::NTuple{N, Int})
```

```
-> <:AbstractDict{typeof(t), <:Number}
```

Perform a braiding of the uncoupled indices of the fusion tree `f` and return the result as a `<:AbstractDict` of output trees and corresponding coefficients. The braiding is determined by specifying that the new sector at position `k` corresponds to the sector that was originally at the position `i = p[k]` , and assigning to every index `i` of the original fusion tree a distinct level or depth `levels[i]` . This permutation is then decomposed into elementary swaps between neighbouring indices, where the swaps are applied as braids such that if `i` and `j` cross, $\tau_{i,j}$ is applied if `levels[i] < levels[j]` and $\tau_{j,i}^{-1}$ if `levels[i] > levels[j]` . This does not allow to encode the most general braid, but a general braid can be obtained by combining such operations.

source

`TensorKit.permute` – Method.



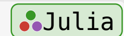
```
permute(f::FusionTree, p::NTuple{N, Int}) ->
<:AbstractDict{typeof(t), <:Number}
```

Perform a permutation of the uncoupled indices of the fusion tree `f` and returns the result as a `<:AbstractDict` of output trees and corresponding coefficients; this requires that `BraidingStyle(sectortype(f)) isa SymmetricBraiding` .

source

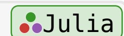
These can be composed to implement elementary manipulations of fusion-splitting tree pairs, according to the following methods

`TensorKit.bendright` – Function.



```
bendright((f₁, f₂)::FusionTreePair) -> (f₃, f₄) => coeff
bendright(src::FusionTreeBlock) -> dst => coeffs
```

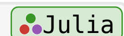
Map the final splitting vertex `a ⊗ b ← c` of `src` to a fusion vertex `a ← c ⊗ dual(b)` in `dst` . For `FusionStyle(src) === UniqueFusion()` , both `src` and `dst` are simple `FusionTreePair` s, and the transformation consists of a single coefficient `coeff` . For generic `FusionStyle` s, the input and output consist of `FusionTreeBlock` s that bundle together all trees with the same uncoupled charges, and `coeffs` now forms a transformation matrix.



See also `bendleft` .

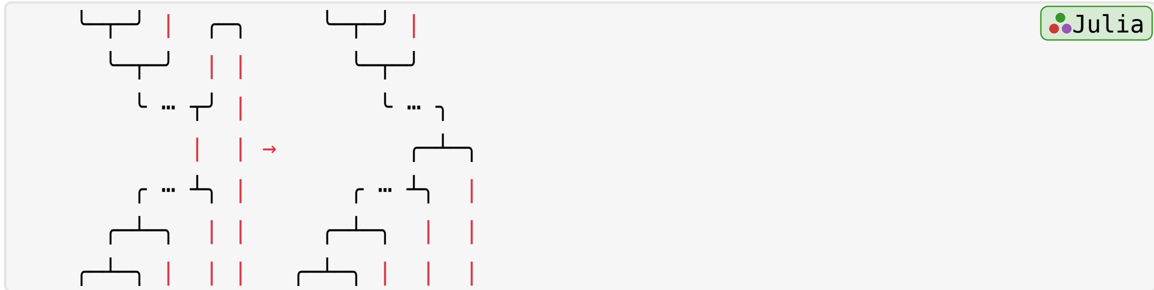
source

`TensorKit.bendleft` – Function.



```
bendleft((f₁, f₂)::FusionTreePair) -> (f₃, f₄) => coeff
bendleft(src::FusionTreeBlock) -> dst => coeffs
```

Map the final fusion vertex `a ← c ⊗ dual(b)` of `src` to a splitting vertex `a ⊗ b ← c` in `dst` . For `FusionStyle(src) === UniqueFusion()` , both `src` and `dst` are simple `FusionTreePair` s, and the transformation consists of a single coefficient `coeff` . For generic `FusionStyle` s, the input and output consist of `FusionTreeBlock` s that bundle together all trees with the same uncoupled charges, and `coeffs` now forms a transformation matrix.



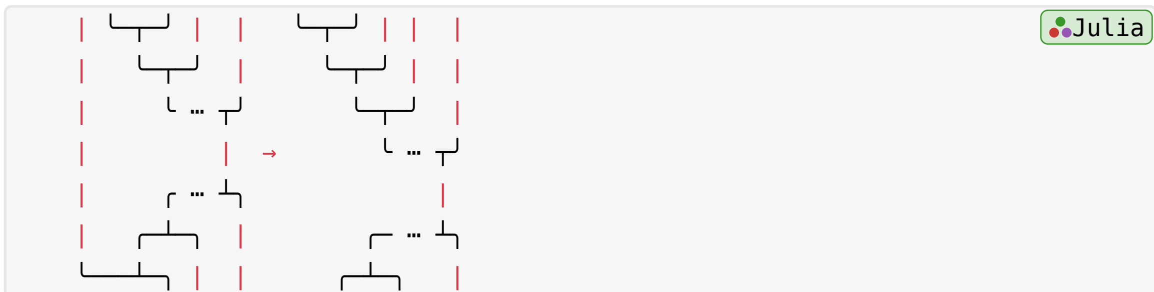
See also `bendright` .

source

`TensorKit.foldright` – Function.

```julia
foldright((f₁, f₂)::FusionTreePair) -> (f₃, f₄) => coeff
foldright(src::FusionTreeBlock) -> dst => coeffs
```

Map the first splitting vertex `a ⊗ b ← c` of `src` to a fusion vertex `b ← dual(a) ⊗ c` , and reexpress as a linear combination of standard basis trees. For `FusionStyle(src) === UniqueFusion()` , both `src` and `dst` are simple `FusionTreePair` s, and the transformation consists of a single coefficient `coeff` . For generic `FusionStyle` s, the input and output consist of `FusionTreeBlock` s that bundle together all trees with the same uncoupled charges, and `coeffs` now forms a transformation matrix.



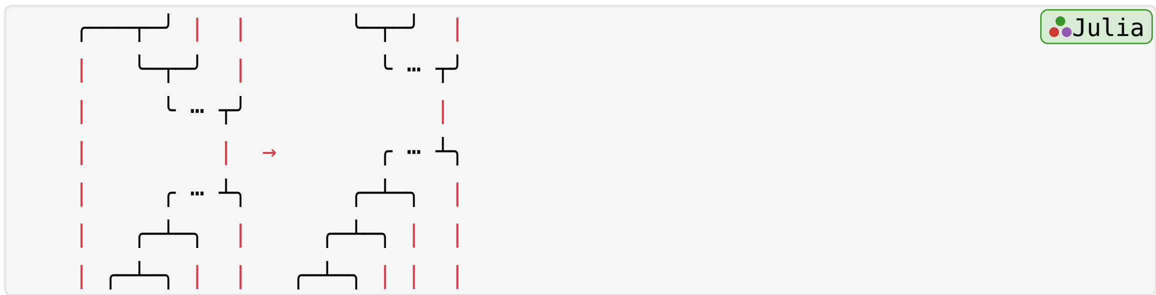
See also `foldleft` .

source

`TensorKit.foldleft` – Function.

```julia
foldleft((f₁, f₂)::FusionTreePair) -> (f₃, f₄) => coeff
foldleft(src::FusionTreeBlock) -> dst => coeffs
```

Map the first fusion vertex `a ← c ⊗ dual(b)` of `src` to a splitting vertex `a ⊗ b ← c` in `dst` . For `FusionStyle(src) === UniqueFusion()` , both `src` and `dst` are simple `FusionTreePair` s, and the transformation consists of a single coefficient `coeff` . For generic `FusionStyle` s, the input and output consist of `FusionTreeBlock` s that bundle together all trees with the same uncoupled charges, and `coeffs` now forms a transformation matrix.



See also `foldright`.

source

Finally, these are used to define large manipulations of fusion-splitting tree pairs, which are then used in the index manipulation of `AbstractTensorMap` objects. The following methods defined on fusion splitting tree pairs have an associated definition for tensors.

`TensorKit.repartition` – Method.

```
repartition((f₁, f₂)::FusionTreePair{I, N₁, N₂}, N::Int) where {I,
N₁, N₂}
    -> <:AbstractDict{<:FusionTreePair{I, N, N₁+N₂-N}}, <:Number}
```

Input is a double fusion tree that describes the fusion of a set of incoming uncoupled sectors to a set of outgoing uncoupled sectors, represented using the individual trees of outgoing (`f₁`) and incoming sectors (`f₂`) respectively (with identical coupled sector `f₁.coupled == f₂.coupled`). Computes new trees and corresponding coefficients obtained from repartitioning the tree by bending incoming to outgoing sectors (or vice versa) in order to have `N` outgoing sectors.

source

`Base.transpose` – Method.

```
transpose((f₁, f₂)::FusionTreePair{I}, p::Index2Tuple{N₁, N₂}) where
{I, N₁, N₂}
    -> <:AbstractDict{<:FusionTreePair{I, N₁, N₂}}, <:Number}
```

Input is a double fusion tree that describes the fusion of a set of incoming uncoupled sectors to a set of outgoing uncoupled sectors, represented using the individual trees of outgoing (`t1`) and incoming sectors (`t2`) respectively (with identical coupled sector `t1.coupled == t2.coupled`). Computes new trees and corresponding coefficients obtained from repartitioning and permuting the tree such that sectors `p1` become outgoing and sectors `p2` become incoming.

source

`TensorKit.braid` – Method.

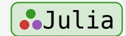
```
braid((f₁, f₂)::FusionTreePair, (p1, p2)::Index2Tuple, (levels1,
levels2)::Index2Tuple)
    -> <:AbstractDict{<:FusionTreePair{I, N₁, N₂}}, <:Number}
```

Input is a fusion-splitting tree pair that describes the fusion of a set of incoming uncoupled sectors to a set of outgoing uncoupled sectors, represented using the splitting tree `f₁` and fusion tree `f₂` , such that the incoming sectors `f₂.uncoupled` are fused to `f₁.coupled == f₂.coupled` and then to the outgoing sectors `f₁.uncoupled` . Compute new trees and corresponding coefficients obtained

from repartitioning and braiding the tree such that sectors `p1` become outgoing and sectors `p2` become incoming. The uncoupled indices in splitting tree `f₁` and fusion tree `f₂` have levels (or depths) `levels1` and `levels2` respectively, which determines how indices braid. In particular, if `i` and `j` cross, $\tau_{i,j}$ is applied if `levels[i] < levels[j]` and $\tau_{j,i}^{-1}$ if `levels[i] > levels[j]`. This does not allow to encode the most general braid, but a general braid can be obtained by combining such operations.

source

`TensorKit.permute` – Method.



```
permute((f₁, f₂)::FusionTreePair, (p1, p2)::Index2Tuple)
    -> <:AbstractDict{<:FusionTreePair{I, N₁, N₂}}, <:Number}
```

Input is a double fusion tree that describes the fusion of a set of incoming uncoupled sectors to a set of outgoing uncoupled sectors, represented using the individual trees of outgoing (`t1`) and incoming sectors (`t2`) respectively (with identical coupled sector `t1.coupled == t2.coupled`). Computes new trees and corresponding coefficients obtained from repartitioning and permuting the tree such that sectors `p1` become outgoing and sectors `p2` become incoming.

source

Chapter 14

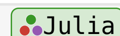
Vector spaces

14.1 Type hierarchy

The following types are defined to characterise vector spaces and their properties:

`TensorKit.Field` – Type.

```
abstract type Field end
```

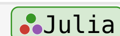


Abstract type at the top of the type hierarchy for denoting fields over which vector spaces (or more generally, linear categories) can be defined. Two common fields are `ℝ` and `ℂ` , representing the field of real or complex numbers respectively.

source

`TensorKit.VectorSpace` – Type.

```
abstract type VectorSpace end
```

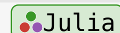


Abstract type at the top of the type hierarchy for denoting vector spaces, or, more generally, objects in linear monoidal categories.

source

`TensorKit.ElementarySpace` – Type.

```
abstract type ElementarySpace <: VectorSpace
```



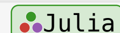
Elementary finite-dimensional vector space over a field that can be used as the index space corresponding to the indices of a tensor. ElementarySpace is a supertype for all vector spaces (objects) that can be associated with the individual indices of a tensor, as hinted to by its alias `IndexSpace` .

Every elementary vector space should respond to the methods `conj` and `dual` , returning the complex conjugate space and the dual space respectively. The complex conjugate of the dual space is obtained as `dual(conj(V)) === conj(dual(V))` . These different spaces should be of the same type, so that a tensor can be defined as an element of a homogeneous tensor product of these spaces.

source

`TensorKit.GeneralSpace` – Type.

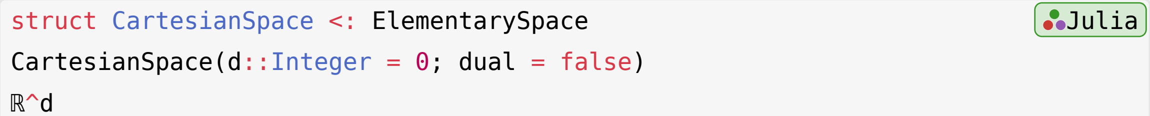
```
struct GeneralSpace{𝔽} <: ElementarySpace
GeneralSpace{𝔽}(d::Integer = 0; dual::Bool = false, conj::Bool = false)
```



A finite-dimensional space over an arbitrary field `𝔽` without additional structure. It is thus characterized by its dimension, and whether or not it is the dual and/or conjugate space. For a real field `𝔽` , the space and its conjugate are the same.

source

`TensorKit.CartesianSpace` – Type.

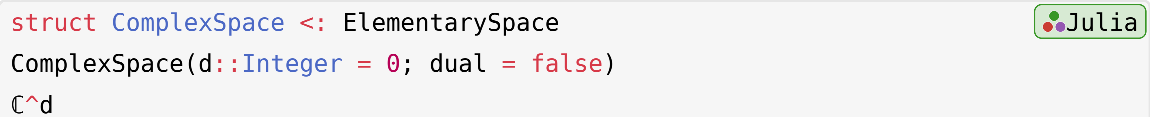


```julia
struct CartesianSpace <: ElementarySpace
CartesianSpace(d::Integer = 0; dual = false)
ℝ^d
```

A real Euclidean space $\mathbb{R}^d$. `CartesianSpace` has no additonal structure and is completely characterised by its dimension `d` . A `dual` keyword argument is accepted for compatibility with other space constructors, but is ignored since the dual of a Cartesian space is isomorphic to itself. This is the vector space that is implicitly assumed in most of matrix algebra.

source

`TensorKit.ComplexSpace` – Type.

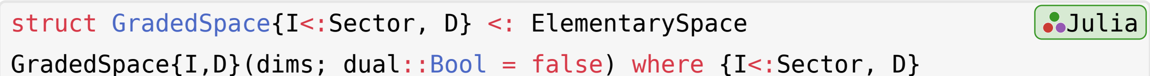


```julia
struct ComplexSpace <: ElementarySpace
ComplexSpace(d::Integer = 0; dual = false)
ℂ^d
```

A standard complex vector space $\mathbb{C}^d$ with Euclidean inner product and no additional structure. It is completely characterised by its dimension and whether its the normal space or its dual (which is canonically isomorphic to the conjugate space).

source

`TensorKit.GradedSpace` – Type.



```julia
struct GradedSpace{I<:Sector, D} <: ElementarySpace
GradedSpace{I,D}(dims; dual::Bool = false) where {I<:Sector, D}
```

A complex Euclidean space with a grading, i.e. a direct sum structure corresponding to labels in a set `I` , the objects of which have the structure of a monoid with respect to a monoidal product `⊗` . In practice, we restrict the label set to be a set of superselection sectors of type `I<:Sector` , e.g. the set of distinct irreps of a finite or compact group, or the isomorphism classes of simple objects of a unitary and pivotal (pre-, multi-) fusion category.

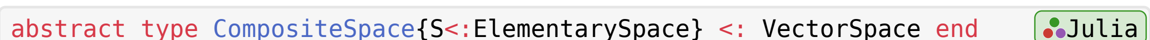
Here `dims` represents the degeneracy or multiplicity of every sector.

The data structure `D` of `dims` will depend on the result `Base.IteratorSize(values(I))` . If the result is of type `HasLength` or `HasShape` , `dims` will be stored in a `NTuple{N,Int}` with `N = length(values(I))` . This requires that a sector `s::I` can be transformed into an index via `s == getindex(values(I), i)` and `i == findindex(values(I), s)` . If `Base.IteratorElsize(values(I))` results `IsInfinite()` or `SizeUnknown()` , a `SectorDict{I,Int}` is used to store the non-zero degeneracy dimensions with the corresponding sector as key. The parameter `D` is hidden from the user and should typically be of no concern.

The concrete type `GradedSpace{I,D}` with correct `D` can be obtained as `Vect[I]` , or if `I == Irrep[G]` for some `G<:Group` , as `Rep[G]` .

source

`TensorKit.CompositeSpace` – Type.

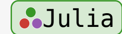


```julia
abstract type CompositeSpace{S<:ElementarySpace} <: VectorSpace end
```

Abstract type for composite spaces that are defined in terms of a number of elementary vector spaces of a homogeneous type `S<:ElementarySpace` .

source

`TensorKit.ProductSpace` – Type.

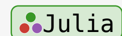


```julia
struct ProductSpace{S <: ElementarySpace, N} <: CompositeSpace{S}
ProductSpace(spaces::NTuple{N, S}) where {S <: ElementarySpace, N}
```

A `ProductSpace` is a tensor product space of `N` vector spaces of type `S <: ElementarySpace` . Only tensor products between `ElementarySpace` objects of the same type are allowed.

source

`TensorKit.HomSpace` – Type.



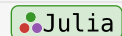
```julia
struct HomSpace{S<:ElementarySpace, P1<:CompositeSpace{S},
P2<:CompositeSpace{S}}
HomSpace(codomain::CompositeSpace{S}, domain::CompositeSpace{S}) where
{S<:ElementarySpace}
```

Represents the linear space of morphisms with codomain of type `P1` and domain of type `P2` . Note that `HomSpace` is not a subtype of `VectorSpace` , i.e. we restrict the latter to denote categories and their objects, and keep `HomSpace` distinct.

source

together with the following specific type for encoding the inner product structure of a space:

`TensorKit.InnerProductStyle` – Type.



```julia
abstract type InnerProductStyle end
InnerProductStyle(V::VectorSpace) -> ::InnerProductStyle
InnerProductStyle(S::Type{<:VectorSpace}) -> ::InnerProductStyle
```

Trait to describe wether vector spaces exhibit an inner product structure, a.k.a. a unitary structure, which can take the following values:

- `EuclideanInnerProduct()` : the metric is the identity, making dual and conjugate spaces equivalent
- `NoInnerProduct()` : no metric and thus no relation between `dual(V)` or `conj(V)`

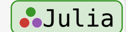
Furthermore, `EuclideanInnerProduct` is a subtype of `HasInnerProduct` , indicating that an inner product exists, and an isomorphism between the dual space and the conjugate space can be constructed. New inner product styles can be defined that subtype `HasInnerProduct` , for example to work with vector spaces with non-trivial metrics. However, at the moment TensorKit does not provide built-in support for such non-standard inner products.

source

14.2 Useful constants

The following constants are defined to easily create the concrete type of `GradedSpace` associated with a given type of sector.

`TensorKit.Vect` – Constant.

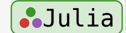


```
const Vect
```

A constant of a singleton type used as `Vect[I]` with `I<:Sector` a type of sector, to construct or obtain the concrete type `GradedSpace{I,D}` instances without having to specify `D` .

source

`TensorKit.Rep` – Constant.



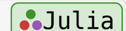
```
const Rep
```

A constant of a singleton type used as `Rep[G]` with `G<:Group` a type of group, to construct or obtain the concrete type `GradedSpace{Irrep[G],D}` instances without having to specify `D` . Note that `Rep[G] == Vect[Irrep[G]]` .

See also `Irrep` and `Vect` .

source

In this respect, there are also a number of type aliases for the `GradedSpace` types associated with the most common sectors, namely

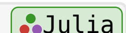


```
const ZNSpace{N} = Vect[ZNIrrep{N}]
const Z2Space = ZNSpace{2}
const Z3Space = ZNSpace{3}
const Z4Space = ZNSpace{4}
const U1Space = Rep[U₁]
const CU1Space = Rep[CU₁]
const SU2Space = Rep[SU₂]

# Unicode alternatives
const ℤ₂Space = Z2Space
const ℤ₃Space = Z3Space
const ℤ₄Space = Z4Space
const U₁Space = U1Space
const CU₁Space = CU1Space
const SU₂Space = SU2Space
```

14.3 Methods

Methods often apply similar to e.g. spaces and corresponding tensors or tensor maps, e.g.:

`TensorKit.field` – Function.

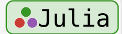


```
field(a) -> Type{𝔽 <: Field}
field(::Type{T}) -> Type{𝔽 <: Field}
```

Return the type of field over which object `a` (e.g. a vector space or a tensor) is defined. This also works in type domain.

source

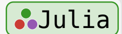
`TensorKit.sectortype` – Function.

```julia
sectortype(a) -> Type{<:Sector}
sectortype(::Type) -> Type{<:Sector}
```


Return the type of sector over which object `a` (e.g. a representation space or a tensor) is defined. Also works in type domain.

source

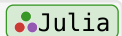
`TensorKit.sectors` – Function.

```julia
sectors(V::ElementarySpace)
```


Return an iterator over the different sectors of `V` .

source

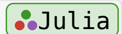
`TensorKit.hassector` – Function.

```julia
hassector(V::VectorSpace, a::Sector) -> Bool
```


Return whether a vector space `V` has a subspace corresponding to sector `a` with non-zero dimension, i.e. `dim(V, a) > 0` .

source

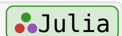
`TensorKitSectors.dim` – Method.

```julia
dim(V::VectorSpace) -> Int
```


Return the total dimension of the vector space `V` as an Int.

source

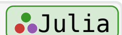
`TensorKitSectors.dim` – Method.

```julia
dim(V::ElementarySpace, s::Sector) -> Int
```


Return the degeneracy dimension corresponding to the sector `s` of the vector space `V` .

source

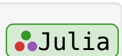
`TensorKit.reduceddim` – Function.

```julia
reduceddim(V::ElementarySpace) -> Int
```


Return the sum of all degeneracy dimensions of the vector space `V` .

source

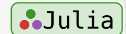
`TensorKitSectors.dim` – Method.

```julia
dim(P::ProductSpace{S, N}, s::NTuple{N, sectortype(S)}) where
{S<:ElementarySpace}
-> Int
```


Return the total degeneracy dimension corresponding to a tuple of sectors for each of the spaces in the tensor product, obtained as `prod(dims(P, s))` `.

source

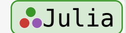
`TensorKitSectors.dim` – Method.

```
dim(W::HomSpace) -> Int
```


Return the total dimension of a `HomSpace` , i.e. the number of linearly independent morphisms that can be constructed within this space.

source

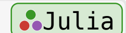
`TensorKit.dims` – Function.

```
dims(::ProductSpace{S, N}) -> Dims{N} = NTuple{N, Int}
dims(V::HomSpace) -> Dims{length(V)}
dims(t::AbstractTensorMap) -> Dims{numind(t)}
```


Return the dimensions of the spaces in the tensor product space(s) as a tuple of integers.

source

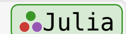
`TensorKit.blocksectors` – Method.

```
blocksectors(P::ProductSpace)
```


Return an iterator over the different unique coupled sector labels, i.e. the different fusion outputs that can be obtained by fusing the sectors present in the different spaces that make up the `ProductSpace` instance.

source

`TensorKit.blocksectors` – Method.

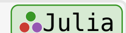
```
blocksectors(W::HomSpace) -> Indices{I}
```


Return an `Indices` of all coupled sectors for `W` . The result is cached based on the sector structure of `W` (ignoring degeneracy dimensions).

See also `hasblock` , `blockstructure` .

source

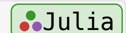
`TensorKit.hasblock` – Function.

```
hasblock(P::ProductSpace, c::Sector)
```


Query whether a coupled sector `c` appears with nonzero dimension in `P` , i.e. whether `blockdim(P, c) > 0` .

See also `blockdim` and `blocksectors` .

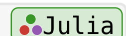
source

```
hasblock(W::HomSpace, c::Sector)
```


Query whether a coupled sector `c` appears in both the codomain and domain of `W` .

See also `blocksectors` .

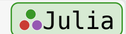
source

```
hasblock(t::AbstractTensorMap, c::Sector) -> Bool
```


Verify whether a tensor has a block corresponding to a coupled sector `c` .

source

`TensorKit.blockdim` – Function.

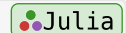
```
blockdim(P::ProductSpace, c::Sector)
```


Return the total dimension of a coupled sector `c` in the product space, by summing over all `dim(P, s)` for all tuples of sectors `s::NTuple{N, Sector}` that can fuse to `c` , counted with the correct multiplicity (i.e. number of ways in which `s` can fuse to `c`).

See also `hasblock` and `blocksectors` .

source

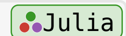
`TensorKit.fusiontrees` – Method.

```
fusiontrees(P::ProductSpace, blocksector::Sector)
```


Return an iterator over all fusion trees that can be formed by fusing the sectors present in the different spaces that make up the `ProductSpace` instance into the coupled sector `blocksector` .

source

`TensorKit.space` – Function.

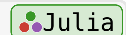
```
space(a) -> VectorSpace
```


Return the vector space associated to object `a` .

source

The following methods act specifically on `ElementarySpace` spaces:

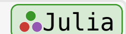
`TensorKitSectors.dual` – Method.

```
dual(V::VectorSpace) -> VectorSpace
```


Return the dual space of `V` ; also obtained via `V'` . This should satisfy `dual(dual(V)) == V` . It is assumed that `typeof(V) == typeof(V')` .

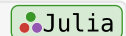
source

`Base.conj` – Function.

```
conj(V::VectorSpace) -> VectorSpace
```


Return the conjugate space of `V` . This should satisfy `conj(conj(V)) == V` . For vector spaces over the real numbers, it must hold that `conj(V) == V` . For vector spaces with a Euclidean inner product, it must hold that `conj(V) == dual(V)` .

source

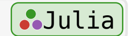
```
conj(V::S) where {S<:ElementarySpace} -> S
```


Return the conjugate space of `V` . This should satisfy `conj(conj(V)) == V` .

For `field(V)==ℝ` , `conj(V) == V` . It is assumed that `typeof(V) == typeof(conj(V))` .

source

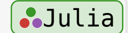
`TensorKit.isconj` – Function.

```
isconj(V::ElementarySpace) -> Bool
```


Return whether an ElementarySpace `V` is normal or rather the conjugated space. Always returns `false` for spaces where `V == conj(V)`, i.e. vector spaces over ℝ.

source

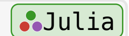
`TensorKit.isdual` – Function.

```
isdual(V::ElementarySpace) -> Bool
```


Return whether an ElementarySpace `V` is normal or rather a dual space. Always returns `false` for spaces where `V == dual(V)`.

source

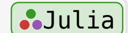
`TensorKit.flip` – Function.

```
flip(V::S) where {S<:ElementarySpace} -> S
```


Return a single vector space of type `S` that has the same value of `isdual` as `dual(V)`, but yet is isomorphic to `V` rather than to `dual(V)`. The spaces `flip(V)` and `dual(V)` only differ in the case of `GradedSpace{I}`.

source

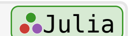
`TensorKit.zerospace` – Function.

```
zerospace(V::S) where {S <: ElementarySpace} -> S
```


Return the corresponding vector space of type `S` that represents the zero-dimensional or empty space. This is the zero element of the direct sum of vector spaces. `Base.zero` falls back to `zerospace`.

source

`TensorKit.unitspace` – Function.

```
unitspace(V::S) where {S <: ElementarySpace} -> S
```


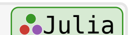
Return the corresponding vector space of type `S` that represents the trivial one-dimensional space, i.e. the space that is isomorphic to the corresponding field. For vector spaces where `I = sectortype(S)` has a semi-simple unit structure (`UnitStyle(I) == GenericUnit()`), this returns a multi-dimensional space corresponding to all unit sectors: `dim(unitspace(V), s) == 1` for all `s in allunits(I)`.

Note

`unitspace(V)` is different from `one(V)`. The latter returns the empty product space `ProductSpace{S,0}(())`. `Base.oneunit` falls back to `unitspace`.

source

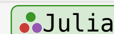
`TensorKit.:⊕` – Function.

```
⊕(V₁::S, V₂::S, V₃::S...) where {S<:ElementarySpace} -> S
oplus(V₁::S, V₂::S, V₃::S...) where {S<:ElementarySpace} -> S
```


Return the corresponding vector space of type `S` that represents the direct sum sum of the spaces `V₁`, `V₂`, ... Note that all the individual spaces should have the same value for `isdual`, as otherwise the direct sum is not defined.

source

`TensorKit.:⊖` – Function.

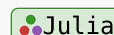


```
⊖(V::ElementarySpace, W::ElementarySpace) -> X::ElementarySpace
ominus(V::ElementarySpace, W::ElementarySpace) -> X::ElementarySpace
```

Return a space that is equivalent to the orthogonal complement of `W` in `V`, i.e. an instance `X::ElementarySpace` such that `V = W ⊕ X`.

source

`TensorKit.supremum` – Function.

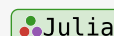


```
supremum(V₁::ElementarySpace, V₂::ElementarySpace,
V₃::ElementarySpace...)
```

Return the supremum of a number of elementary spaces, i.e. an instance `V::ElementarySpace` such that `V ≿ V₁`, `V ≿ V₂`, ... and no other `W ≺ V` has this property. This requires that all arguments have the same value of `isdual( )`, and also the return value `V` will have the same value.

source

`TensorKit.infimum` – Function.



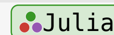
```
infimum(V₁::ElementarySpace, V₂::ElementarySpace,
V₃::ElementarySpace...)
```

Return the infimum of a number of elementary spaces, i.e. an instance `V::ElementarySpace` such that `V ≾ V₁`, `V ≾ V₂`, ... and no other `W ≻ V` has this property. This requires that all arguments have the same value of `isdual( )`, and also the return value `V` will have the same value.

source

while the following also work on both `ElementarySpace` and `ProductSpace`

`TensorKit.fuse` – Function.

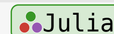


```
fuse(V₁::S, V₂::S, V₃::S...) where {S<:ElementarySpace} -> S
fuse(P::ProductSpace{S}) where {S<:ElementarySpace} -> S
```

Return a single vector space of type `S` that is isomorphic to the fusion product of the individual spaces `V₁`, `V₂`, ..., or the spaces contained in `P`.

source

`Base.one` – Method.

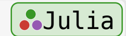


```
one(::S) where {S <: ElementarySpace} -> ProductSpace{S, 0}
one(::ProductSpace{S}) where {S <: ElementarySpace} -> ProductSpace{S, 0}
```

Return a tensor product of zero spaces of type `S`, i.e. this is the unit object under the tensor product operation, such that `V ⊗ one(V) == V`.

source

`TensorKitSectors.:⊗` – Method.



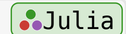
```
⊗(V₁::S, V₂::S, V₃::S...) where {S<:ElementarySpace} -> S
```

Create a `ProductSpace{S}(V₁, V₂, V₃...)` representing the tensor product of several elementary vector spaces. For convience, Julia's regular multiplication operator * applied to vector spaces has the same effect.

The tensor product structure is preserved, see `fuse` for returning a single elementary space of type `S` that is isomorphic to this tensor product.

source

`TensorKitSectors.:⊠` – Method.



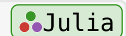
```
⊠(V₁::VectorSpace, V₂::VectorSpace)
```

Given two vector spaces `V₁` and `V₂` (`ElementarySpace` or `ProductSpace`), or thus, objects of corresponding fusion categories C_1 and C_2, $V_1 \boxtimes V_2$ constructs the Deligne tensor product, an object in $C_1 \boxtimes C_2$ which is the natural tensor product of those categories. In particular, the corresponding type of sectors (simple objects) is given by `sectortype(V₁ ⊠ V₂) == sectortype(V₁) ⊠ sectortype(V₂)` and can be thought of as a tuple of the individual sectors.

The Deligne tensor product also works in the type domain and for sectors and tensors. For group representations, we have `Rep[G₁] ⊠ Rep[G₂] == Rep[G₁ × G₂]` , i.e. these are the natural representation spaces of the direct product of two groups.

source

`TensorKit.ismonomorphic` – Function.

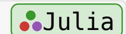


```
ismonomorphic(V₁::VectorSpace, V₂::VectorSpace)
V₁ ≾ V₂
```

Return whether there exist monomorphisms from `V₁` to `V₂` , i.e. 'injective' morphisms with left inverses.

source

`TensorKit.isepimorphic` – Function.

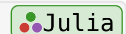


```
isepimorphic(V₁::VectorSpace, V₂::VectorSpace)
V₁ ≿ V₂
```

Return whether there exist epimorphisms from `V₁` to `V₂` , i.e. 'surjective' morphisms with right inverses.

source

`TensorKit.isisomorphic` – Function.



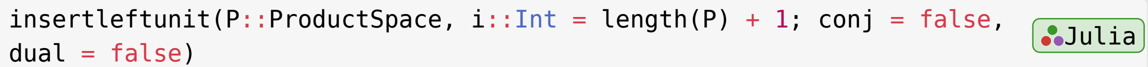
```
isisomorphic(V₁::VectorSpace, V₂::VectorSpace)
V₁ ≅ V₂
```

Return if `V₁` and `V₂` are isomorphic, meaning that there exists isomorphisms from `V₁` to `V₂` , i.e. morphisms with left and right inverses.

source

Inserting trivial space factors or removing such factors for `ProductSpace` instances can be done with the following methods.

`TensorKit.insertleftunit` – Method.



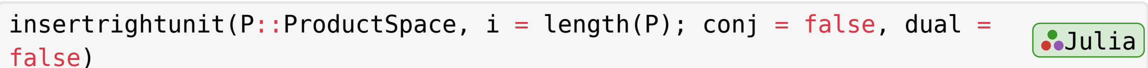
```julia
insertleftunit(P::ProductSpace, i::Int = length(P) + 1; conj = false,
dual = false)
```

Insert a trivial vector space, isomorphic to the underlying field, at position `i`, which can be specified as an `Int` or as `Val(i)` for improved type stability. More specifically, adds a left monoidal unit or its dual.

See also `insertrightunit`, `removeunit`.

source

`TensorKit.insertrightunit` – Method.



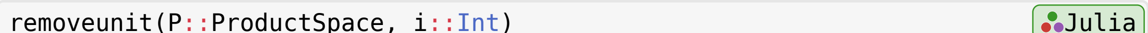
```julia
insertrightunit(P::ProductSpace, i = length(P); conj = false, dual =
false)
```

Insert a trivial vector space, isomorphic to the underlying field, after position `i`, which can be specified as an `Int` or as `Val(i)` for improved type stability. More specifically, adds a right monoidal unit or its dual.

See also `insertleftunit`, `removeunit`.

source

`TensorKit.removeunit` – Method.



```julia
removeunit(P::ProductSpace, i::Int)
```

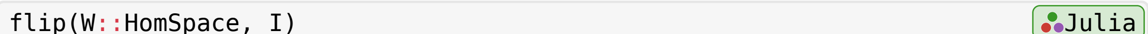
This removes a trivial tensor product factor at position `1 ≤ i ≤ N`, where `i` can be specified as an `Int` or as `Val(i)` for improved type stability. For this to work, that factor has to be isomorphic to the field of scalars.

This operation undoes the work of `insertleftunit` and `insertrightunit`.

source

There are also specific methods for `HomSpace` instances, that are used in determining the resulting `HomSpace` after applying certain tensor operations.

`TensorKit.flip` – Method.



```julia
flip(W::HomSpace, I)
```

Return a new `HomSpace` object by applying `flip` to each of the spaces in the domain and codomain of `W` for which the linear index `i` satisfies `i ∈ I`.

source

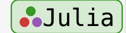
`TensorKit.permute` – Method.

```julia
permute(W::HomSpace, (p₁, p₂)::Index2Tuple)
```

Return the `HomSpace` obtained by permuting the indices of the domain and codomain of `W` according to the permutation p_1 and p_2 respectively.

source

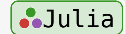
TensorKit.select – Method.

```
select(W::HomSpace, (p₁, p₂)::Index2Tuple{N₁,N₂})
```


Return the HomSpace obtained by a selection from the domain and codomain of W according to the indices in p_1 and p_2 respectively.

source

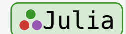
TensorKit.compose – Method.

```
compose(W::HomSpace, V::HomSpace)
```


Obtain the HomSpace that is obtained from composing the morphisms in W and V . For this to be possible, the domain of W must match the codomain of V .

source

TensorKit.insertleftunit – Method.

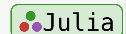
```
insertleftunit(W::HomSpace, i = numind(W) + 1; conj = false, dual =
false)
```


Insert a trivial vector space, isomorphic to the underlying field, at position i , which can be specified as an Int or as Val(i) for improved type stability. More specifically, adds a left monoidal unit or its dual.

See also insertrightunit , removeunit .

source

TensorKit.insertrightunit – Method.

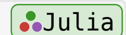
```
insertrightunit(W::HomSpace, i = numind(W); conj = false, dual =
false)
```


Insert a trivial vector space, isomorphic to the underlying field, after position i , which can be specified as an Int or as Val(i) for improved type stability. More specifically, adds a right monoidal unit or its dual.

See also insertleftunit , removeunit .

source

TensorKit.removeunit – Method.

```
removeunit(P::HomSpace, i)
```


This removes a trivial tensor product factor at position 1 ≤ i ≤ N , where i can be specified as an Int or as Val(i) for improved type stability. For this to work, the space at position i has to be isomorphic to the field of scalars.

This operation undoes the work of insertleftunit and insertrightunit .

source

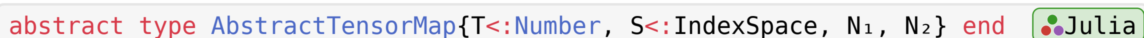
Chapter 15

Tensors

15.1 Type hierarchy

The abstract supertype of all tensors in TensorKit is given by `AbstractTensorMap` :

`TensorKit.AbstractTensorMap` – Type.



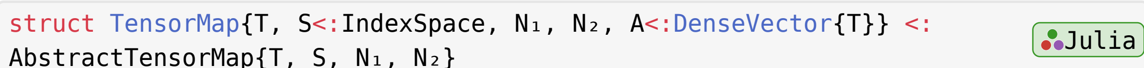
```julia
abstract type AbstractTensorMap{T<:Number, S<:IndexSpace, N₁, N₂} end
```

Abstract supertype of all tensor maps, i.e. linear maps between tensor products of vector spaces of type `S<:IndexSpace` , with element type `T` . An `AbstractTensorMap` maps from an input space of type `ProductSpace{S, N₂}` to an output space of type `ProductSpace{S, N₁}` .

source

The following concrete subtypes are provided within the TensorKit library:

`TensorKit.TensorMap` – Type.

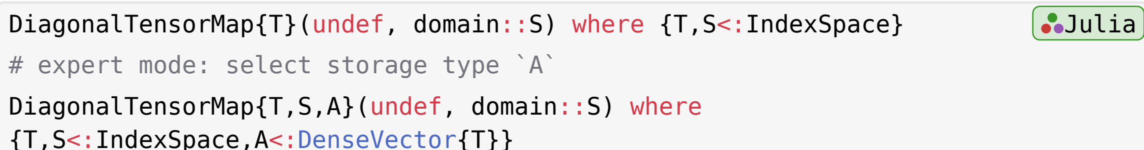


```julia
struct TensorMap{T, S<:IndexSpace, N₁, N₂, A<:DenseVector{T}} <:
AbstractTensorMap{T, S, N₁, N₂}
```

Specific subtype of `AbstractTensorMap` for representing tensor maps (morphisms in a tensor category), where the data is stored in a dense vector.

source

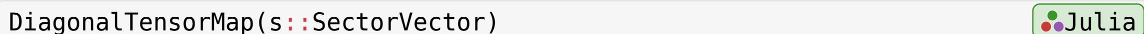
`TensorKit.DiagonalTensorMap` – Type.



```julia
DiagonalTensorMap{T}(undef, domain::S) where {T,S<:IndexSpace}
# expert mode: select storage type `A`
DiagonalTensorMap{T,S,A}(undef, domain::S) where
{T,S<:IndexSpace,A<:DenseVector{T}}
```

Construct a `DiagonalTensorMap` with uninitialized data.

source

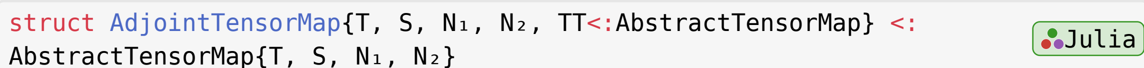


```julia
DiagonalTensorMap(s::SectorVector)
```

Construct a `DiagonalTensorMap` directly from a `SectorVector` , from which the codomain (assumed non-dual) is inferred automatically.

source

`TensorKit.AdjointTensorMap` – Type.

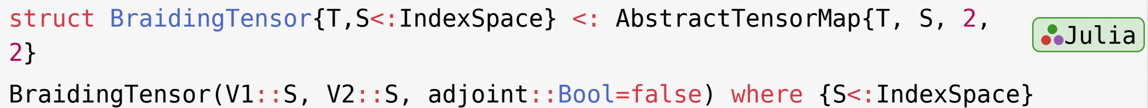


```julia
struct AdjointTensorMap{T, S, N₁, N₂, TT<:AbstractTensorMap} <:
AbstractTensorMap{T, S, N₁, N₂}
```

Specific subtype of `AbstractTensorMap` that is a lazy wrapper for representing the adjoint of an instance of `AbstractTensorMap` .

source

`TensorKit.BraidingTensor` – Type.



```julia
struct BraidingTensor{T,S<:IndexSpace} <: AbstractTensorMap{T, S, 2,
2}
BraidingTensor(V1::S, V2::S, adjoint::Bool=false) where {S<:IndexSpace}
```

Specific subtype of `AbstractTensorMap` for representing the braiding tensor that braids the first input over the second input; its inverse can be obtained as the adjoint.

It holds that `domain(BraidingTensor(V1, V2)) == V1 ⊗ V2` and `codomain(BraidingTensor(V1, V2)) == V2 ⊗ V1` .

source

Of those, `TensorMap` provides the generic instantiation of our tensor concept. It supports various constructors, which are discussed in the next subsection.

Furthermore, some aliases are provided for convenience:

`TensorKit.AbstractTensor` – Type.

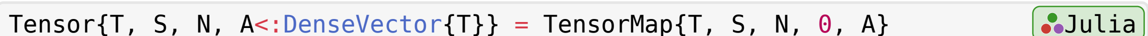
```julia
AbstractTensor{T,S,N} = AbstractTensorMap{T,S,N,0}
```

Abstract supertype of all tensors, i.e. elements in the tensor product space of type `ProductSpace{S, N}` , with element type `T` .

An `AbstractTensor{T, S, N}` is actually a special case `AbstractTensorMap{T, S, N, 0}` , i.e. a tensor map with only non-trivial output spaces.

source

`TensorKit.Tensor` – Type.



```julia
Tensor{T, S, N, A<:DenseVector{T}} = TensorMap{T, S, N, 0, A}
```

Specific subtype of `AbstractTensor` for representing tensors whose data is stored in a dense vector.

A `Tensor{T, S, N, A}` is actually a special case `TensorMap{T, S, N, 0, A}` , i.e. a tensor map with only a non-trivial output space.

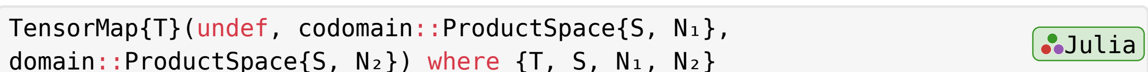
source

15.2 TensorMap constructors

General constructors

A `TensorMap` with undefined data can be constructed by specifying its domain and codomain:

`TensorKit.TensorMap` – Method.



```julia
TensorMap{T}(undef, codomain::ProductSpace{S, N₁},
domain::ProductSpace{S, N₂}) where {T, S, N₁, N₂}
```

```
TensorMap{T}(undef, codomain ← domain)
TensorMap{T}(undef, domain → codomain)
```

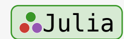
Construct a `TensorMap` with uninitialized data with elements of type `T` .

source

The resulting object can then be filled with data using the `setindex!` method as discussed below, using functions such as `VectorInterface.zerovector!` , `rand!` or `fill!` , or it can be used as an output argument in one of the many methods that accept output arguments, or in an `@tensor output[...] = ...` expression.

Alternatively, a `TensorMap` can be constructed by specifying its data, codomain and domain in one of the following ways:

`TensorKit.TensorMap` – Method.



```
TensorMap(data::AbstractDict{<:Sector, <:AbstractMatrix},
codomain::ProductSpace, domain::ProductSpace)
TensorMap(data, codomain ← domain)
TensorMap(data, domain → codomain)
```

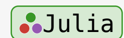
Construct a `TensorMap` by explicitly specifying its block data.

Arguments

- `data::AbstractDict{<:Sector, <:AbstractMatrix}` : dictionary containing the block data for each coupled sector `c` as a matrix of size `(blockdim(codomain, c), blockdim(domain, c))` .
- `codomain::ProductSpace{S, N₁}` : the codomain as a `ProductSpace` of `N₁` spaces of type `S <: ElementarySpace` .
- `domain::ProductSpace{S, N₂}` : the domain as a `ProductSpace` of `N₂` spaces of type `S <: ElementarySpace` .

source

`TensorKit.TensorMap` – Method.



```
TensorMap(data::AbstractArray, codomain::ProductSpace{S,N₁},
domain::ProductSpace{S,N₂};
                tol=sqrt(eps(real(float(eltype(data)))))) where
                {S<:ElementarySpace,N₁,N₂}
TensorMap(data, codomain ← domain; tol=sqrt(eps(real(float(eltype(data))))))
TensorMap(data, domain → codomain; tol=sqrt(eps(real(float(eltype(data))))))
```

Construct a `TensorMap` from a plain multidimensional array.

Arguments

- `data::DenseArray` : tensor data as a plain array.
- `codomain::ProductSpace{S,N₁}` : the codomain as a `ProductSpace` of `N₁` spaces of type `S<:ElementarySpace` .
- `domain::ProductSpace{S,N₂}` : the domain as a `ProductSpace` of `N₂` spaces of type `S<:ElementarySpace` .
- `tol=sqrt(eps(real(float(eltype(data)))))::Float64` :

Here, `data` can be specified in three ways:

1. `data` can be a `DenseVector` of length `dim(codomain ← domain)` ; in that case it represents the actual independent entries of the tensor map. An instance will be created that directly references `data` .
2. `data` can be an `AbstractMatrix` of size `(dim(codomain), dim(domain))`
3. `data` can be an `AbstractArray` of rank `N₁ + N₂` with a size matching that of the domain and codomain spaces, i.e. `size(data) == (dims(codomain)..., dims(domain)...)`

In cases 2 and 3, the `TensorMap` constructor will reconstruct the tensor data such that the resulting tensor `t` satisfies `data == convert(Array, t)` , up to an error specified by `tol` . For the case where `sectortype(S) == Trivial` and `data isa DenseArray` , the `data` array is simply reshaped into a vector and used as in case 1 so that the memory will still be shared. In other cases, new memory will be allocated.

Note that in the case of `N₁ + N₂ = 1` , case 3 also amounts to `data` being a vector, whereas when `N₁ + N₂ == 2` , case 2 and case 3 both require `data` to be a matrix. Such ambiguous cases are resolved by checking the size of `data` in an attempt to support all possible cases.

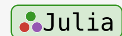
Note

This constructor for case 2 and 3 only works for `sectortype` values for which conversion to a plain array is possible, and only in the case where the `data` actually respects the specified symmetry structure, up to a tolerance `tol` .

source

Finally, we also support the following `Array` -like constructors

`Base.zeros` – Method.

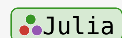


```
zeros([T=Float64,], codomain::ProductSpace{S,N₁},
domain::ProductSpace{S,N₂}) where {S,N₁,N₂,T}
zeros([T=Float64,], codomain ← domain)
```

Create a `TensorMap` with element type `T` , of all zeros with spaces specified by `codomain` and `domain` .

source

`Base.ones` – Method.

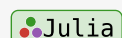


```
ones([T=Float64,], codomain::ProductSpace{S,N₁},
domain::ProductSpace{S,N₂}) where {S,N₁,N₂,T}
ones([T=Float64,], codomain ← domain)
```

Create a `TensorMap` with element type `T` , of all ones with spaces specified by `codomain` and `domain` .

source

`Base.rand` – Method.



```
rand([rng=default_rng()], [TorA=Float64],
codomain::ProductSpace{S,N₁},
             domain::ProductSpace{S,N₂}) where {S,N₁,N₂,T} -> t
```

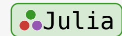
```
rand([rng=default_rng()], [TorA=Float64], codomain ← domain) -> t
```

Generate a tensor `t` with entries generated by `rand` . The type `TorA` can be used to control the element type and data type generated. For example, if `TorA` is a `CuVector{ComplexF32}` or `ROCVector{Float64}` , then the final output `TensorMap` will have that as its storage type.

See also `Random.rand!` .

source

`Base.randn` – Method.



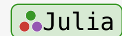
```
randn([rng=default_rng()], [TorA=Float64],
codomain::ProductSpace{S,N₁},
             domain::ProductSpace{S,N₂}) where {S,N₁,N₂,T} -> t
randn([rng=default_rng()], [TorA=Float64], codomain ← domain) -> t
```

Generate a tensor `t` with entries generated by `randn` . The type `TorA` can be used to control the element type and data type generated. For example, if `TorA` is a `CuVector{ComplexF32}` or `ROCVector{Float64}` , then the final output `TensorMap` will have that as its storage type.

See also `Random.randn!` .

source

`Random.randexp` – Method.



```
randexp([rng=default_rng()], [TorA=Float64],
codomain::ProductSpace{S,N₁},
             domain::ProductSpace{S,N₂}) where {S,N₁,N₂,T} -> t
randexp([rng=default_rng()], [TorA=Float64], codomain ← domain) -> t
```

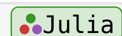
Generate a tensor `t` with entries generated by `randexp` . The type `TorA` can be used to control the element type and data type generated. For example, if `TorA` is a `CuVector{ComplexF32}` or `ROCVector{Float64}` , then the final output `TensorMap` will have that as its storage type.

See also `Random.randexp!` .

source

as well as a `similar` constructor

`Base.similar` – Method.



```
similar(t::AbstractTensorMap, [AorT=storagetype(t)], [V=space(t)])
similar(t::AbstractTensorMap, [AorT=storagetype(t)], codomain, domain)
```

Creates an uninitialized mutable tensor with the given scalar or storagetype `AorT` and structure `V` or `codomain ← domain` , based on the source tensormap. The second and third arguments are both optional, defaulting to the given tensor's `storagetype` and `space` . The structure may be specified either as a single `HomSpace` argument or as `codomain` and `domain` .

By default, this will result in `TensorMap{T}(undef, V)` when custom objects do not specialize this method.

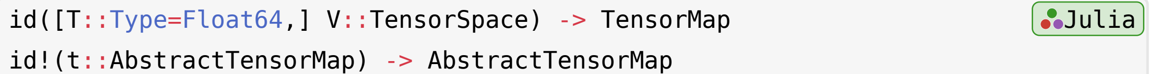
See also `similar_diagonal` .

source

Specific constructors

Additionally, the following methods can be used to construct specific `TensorMap` instances.

`TensorKit.id` – Function.

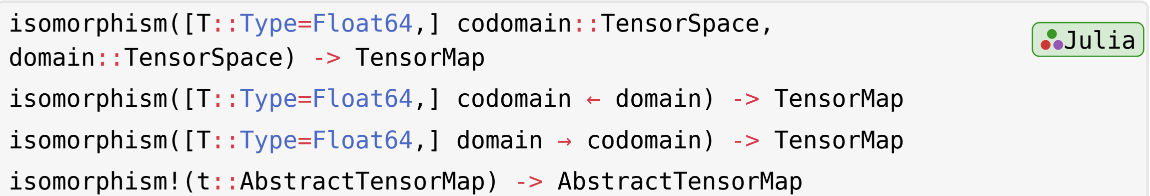


```julia
id([T::Type=Float64,] V::TensorSpace) -> TensorMap
id!(t::AbstractTensorMap) -> AbstractTensorMap
```

Construct the identity endomorphism on space `V`, i.e. return a `t::TensorMap` with `domain(t) == codomain(t) == V`, where either `scalartype(t) = T` if `T` is a `Number` type or `storagetype(t) = T` if `T` is a `DenseVector` type.

source

`TensorKit.isomorphism` – Function.



```julia
isomorphism([T::Type=Float64,] codomain::TensorSpace,
domain::TensorSpace) -> TensorMap
isomorphism([T::Type=Float64,] codomain ← domain) -> TensorMap
isomorphism([T::Type=Float64,] domain → codomain) -> TensorMap
isomorphism!(t::AbstractTensorMap) -> AbstractTensorMap
```

Construct a specific isomorphism between the codomain and the domain, i.e. return a `t::TensorMap` where either `scalartype(t) = T` if `T` is a `Number` type or `storagetype(t) = T` if `T` is a `DenseVector` type. If the spaces are not isomorphic, an error will be thrown.

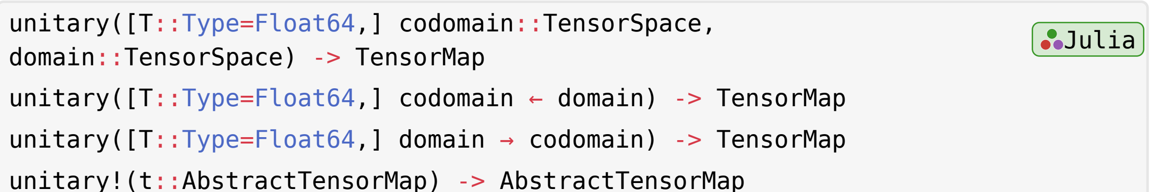
Note

There is no canonical choice for a specific isomorphism, but the current choice is such that `isomorphism(cod, dom) == inv(isomorphism(dom, cod))`.

See also `unitary` when `InnerProductStyle(cod) === EuclideanInnerProduct()`.

source

`TensorKit.unitary` – Function.



```julia
unitary([T::Type=Float64,] codomain::TensorSpace,
domain::TensorSpace) -> TensorMap
unitary([T::Type=Float64,] codomain ← domain) -> TensorMap
unitary([T::Type=Float64,] domain → codomain) -> TensorMap
unitary!(t::AbstractTensorMap) -> AbstractTensorMap
```

Construct a specific unitary morphism between the codomain and the domain, i.e. return a `t::TensorMap` where either `scalartype(t) = T` if `T` is a `Number` type or `storagetype(t) = T` if `T` is a `DenseVector` type. If the spaces are not isomorphic, or the spacetype does not have a Euclidean inner product, an error will be thrown.

Note

There is no canonical choice for a specific unitary, but the current choice is such that `unitary(cod, dom) == inv(unitary(dom, cod)) = adjoint(unitary(dom, cod))`.

See also `isomorphism` and `isometry`.

source

`TensorKit.isometry` – Function.

```julia
isometry([T::Type=Float64,] codomain::TensorSpace,
domain::TensorSpace) -> TensorMap
isometry([T::Type=Float64,] codomain ← domain) -> TensorMap
isometry([T::Type=Float64,] domain → codomain) -> TensorMap
isometry!(t::AbstractTensorMap) -> AbstractTensorMap
```

Construct a specific isometry between the codomain and the domain, i.e. return a `t::TensorMap` where either `scalartype(t) = T` if `T` is a `Number` type or `storagetype(t) = T` if `T` is a `DenseVector` type. The isometry `t` then satisfies `t' * t = id(domain)` and `(t * t')^2 = t * t'`. If the spaces do not allow for such an isometric inclusion, an error will be thrown.

See also `isomorphism` and `unitary`.

source

15.3 `AbstractTensorMap` properties and data access

The following methods exist to obtain type information:

`Base.eltype` – Method.

```julia
eltype(::AbstractTensorMap) -> Type{T}
eltype(::Type{<:AbstractTensorMap}) -> Type{T}
```

Return the scalar or element type `T` of a tensor.

source

`TensorKit.spacetype` – Method.

```julia
spacetype(a) -> Type{S <: IndexSpace}
spacetype(::Type) -> Type{S <: IndexSpace}
```

Return the type of the elementary space `S` of object `a` (e.g. a tensor). Also works in type domain.

source

`TensorKit.sectortype` – Method.

```julia
sectortype(a) -> Type{<:Sector}
sectortype(::Type) -> Type{<:Sector}
```

Return the type of sector over which object `a` (e.g. a representation space or a tensor) is defined. Also works in type domain.

source

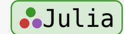
`TensorKit.field` – Method.

```julia
field(a) -> Type{𝔽 <: Field}
field(::Type{T}) -> Type{𝔽 <: Field}
```

Return the type of field over which object `a` (e.g. a vector space or a tensor) is defined. This also works in type domain.

source

`TensorKit.storagetype` – Function.



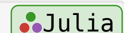
```
storagetype(t::AbstractTensorMap) -> Type{A<:AbstractVector}
storagetype(T::Type{<:AbstractTensorMap}) -> Type{A<:AbstractVector}
```

Return the type of vector that stores the data of a tensor. If this is not overloaded for a given tensor type, the default value of `storagetype(scalartype(t))` is returned.

See also `similarstoragetype` .

source

`TensorKit.blocktype` – Function.



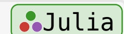
```
blocktype(t)
```

Return the type of the matrix blocks of a tensor.

source

To obtain information about the indices, you can use:

`TensorKit.space` – Method.

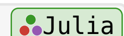


```
space(t::AbstractTensorMap{T,S,N₁,N₂}) -> HomSpace{S,N₁,N₂}
space(t::AbstractTensorMap{T,S,N₁,N₂}, i::Int) -> S
```

The index information of a tensor, i.e. the `HomSpace` of its domain and codomain. If `i` is specified, return the `i` -th index space.

source

`TensorKit.domain` – Function.



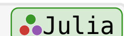
```
domain(t::AbstractTensorMap{T,S,N₁,N₂}) -> ProductSpace{S,N₂}
domain(t::AbstractTensorMap{T,S,N₁,N₂}, i::Int) -> S
```

Return the domain of a tensor, i.e. the product space of the input spaces. If `i` is specified, return the `i` -th input space. Implementations should provide `domain(t)` .

See also `codomain` and `space` .

source

`TensorKit.codomain` – Function.



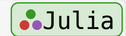
```
codomain(t::AbstractTensorMap{T,S,N₁,N₂}) -> ProductSpace{S,N₁}
codomain(t::AbstractTensorMap{T,S,N₁,N₂}, i::Int) -> S
```

Return the codomain of a tensor, i.e. the product space of the output spaces. If `i` is specified, return the `i` -th output space. Implementations should provide `codomain(t)` .

See also `domain` and `space` .

source

`TensorKit.numin` – Function.



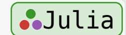
```
numin(x) -> Int
numin(T::Type) -> Int
```

Return the length of the domain, i.e. the number of input spaces. By default, this is implemented in the type domain.

See also `numout` and `numind` .

source

`TensorKit.numout` – Function.



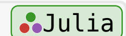
```
numout(x) -> Int
numout(T::Type) -> Int
```

Return the length of the codomain, i.e. the number of output spaces. By default, this is implemented in the type domain.

See also `numin` and `numind` .

source

`TensorKit.numind` – Function.



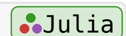
```
numind(x) -> Int
numind(T::Type) -> Int
order(x) = numind(x)
```

Return the total number of input and output spaces, i.e. `numin(x) + numout(x)` . Alternatively, the alias `order` can also be used.

See also `numout` and `numin` .

source

`TensorKit.codomainind` – Function.



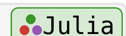
```
codomainind(x) -> Tuple{Int}
```

Return all indices of the codomain.

See also `domainind` and `allind` .

source

`TensorKit.domainind` – Function.



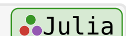
```
domainind(x) -> Tuple{Int}
```

Return all indices of the domain.

See also `codomainind` and `allind` .

source

`TensorKit.allind` – Function.



```
allind(x) -> Tuple{Int}
```

Return all indices, i.e. the indices of both domain and codomain.

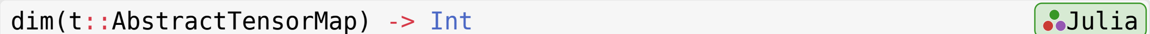
See also `codomainind` and `domainind` .

source

In `TensorMap` instances, all data is gathered in a single `AbstractVector` , which has an internal structure into blocks associated to total coupled charge, within which live subblocks associated with the different possible fusion-splitting tree pairs.

To obtain information about the structure of the data, you can use:

`TensorKitSectors.dim` – Method.

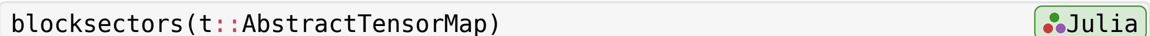


```
dim(t::AbstractTensorMap) -> Int
```

The total number of free parameters of a tensor, discounting the entries that are fixed by symmetry. This is also the dimension of the `HomSpace` on which the `TensorMap` is defined.

source

`TensorKit.blocksectors` – Method.

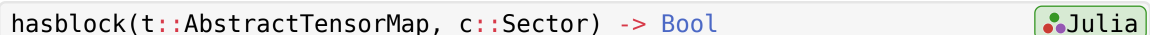


```
blocksectors(t::AbstractTensorMap)
```

Return an iterator over all coupled sectors of a tensor.

source

`TensorKit.hasblock` – Method.

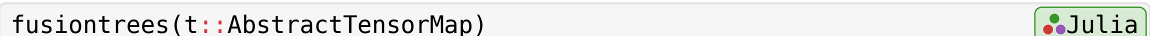


```
hasblock(t::AbstractTensorMap, c::Sector) -> Bool
```

Verify whether a tensor has a block corresponding to a coupled sector `c` .

source

`TensorKit.fusiontrees` – Method.



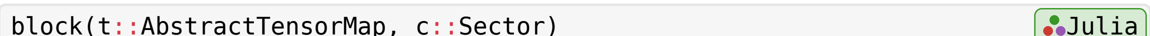
```
fusiontrees(t::AbstractTensorMap)
```

Return an iterator over all splitting - fusion tree pairs of a tensor.

source

Data can be accessed (and modified) in a number of ways. To access the full matrix block associated with the coupled charges, you can use:

`TensorKit.block` – Function.



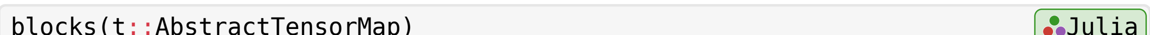
```
block(t::AbstractTensorMap, c::Sector)
```

Return the matrix block of a tensor corresponding to a coupled sector `c` .

See also `blocks` , `blocksectors` , `blockdim` and `hasblock` .

source

`TensorKit.blocks` – Function.



```
blocks(t::AbstractTensorMap)
```

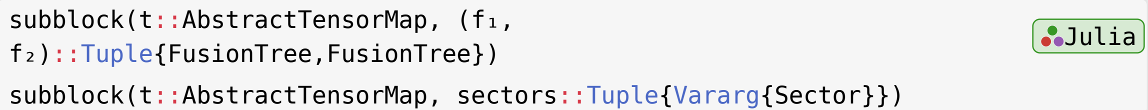
Return an iterator over all blocks of a tensor, i.e. all coupled sectors and their corresponding matrix blocks.

See also `block` , `blocksectors` , `blockdim` and `hasblock` .

source

To access the reduced tensor elements associated to fusion tree pairs, you can use:

`TensorKit.subblock` – Function.



```julia
subblock(t::AbstractTensorMap, (f₁, f₂)::Tuple{FusionTree,FusionTree})
subblock(t::AbstractTensorMap, sectors::Tuple{Vararg{Sector}})
```

Return a view into the data of `t` corresponding to the splitting - fusion tree pair `(f₁, f₂)` . In particular, this is an `AbstractArray{T}` with `T = scalartype(t)` , of size `(dims(codomain(t), f₁.uncoupled)..., dims(codomain(t), f₂.uncoupled)...)` .

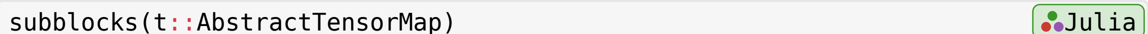
Whenever `FusionStyle(sectortype(t)) isa UniqueFusion` , it is also possible to provide only the external `sectors` , in which case the fusion tree pair will be constructed automatically.

In general, new tensor types should provide an implementation of this function for the fusion tree signature.

See also `subblocks` and `fusiontrees` .

source

`TensorKit.subblocks` – Function.



```julia
subblocks(t::AbstractTensorMap)
```

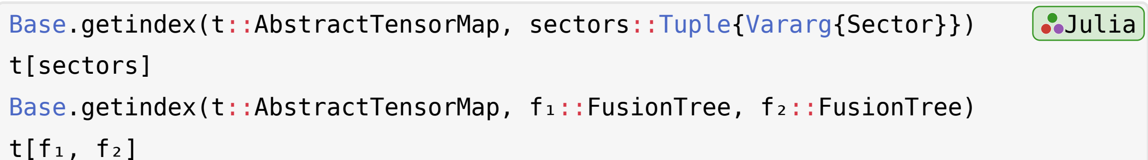
Return an iterator over all subblocks of a tensor, i.e. all fusiontrees and their corresponding tensor subblocks.

See also `subblock` , `fusiontrees` , and `hassubblock` .

source

To access the data associated with a specific fusion tree pair, you can use:

`Base.getindex` – Method.



```julia
Base.getindex(t::AbstractTensorMap, sectors::Tuple{Vararg{Sector}})
t[sectors]
Base.getindex(t::AbstractTensorMap, f₁::FusionTree, f₂::FusionTree)
t[f₁, f₂]
```

Return a view into the data of `t` corresponding to the splitting - fusion tree pair `(f₁, f₂)` . In particular, this is an `AbstractArray{T}` with `T = scalartype(t)` , of size `(dims(codomain(t), f₁.uncoupled)..., dims(codomain(t), f₂.uncoupled)...)` .

Whenever `FusionStyle(sectortype(t)) isa UniqueFusion` , it is also possible to provide only the external `sectors` , in which case the fusion tree pair will be constructed automatically.

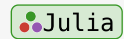
Warning

Contrary to Julia's array types, the default behavior is to return a view into the tensor data. As a result, modifying the view will modify the data in the tensor.

See also `subblock` , `subblocks` and `fusiontrees` .

source

`Base.setindex!` – Method.



```
Base.setindex!(t::AbstractTensorMap, v,
sectors::Tuple{Vararg{Sector}})
t[sectors] = v
Base.setindex!(t::AbstractTensorMap, v, f₁::FusionTree, f₂::FusionTree)
t[f₁, f₂] = v
```

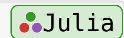
Copies `v` into the data slice of `t` corresponding to the splitting - fusion tree pair `(f₁, f₂)` . By default, `v` can be any object that can be copied into the view associated with `t[f₁, f₂]` .

See also `subblock` , `subblocks` and `fusiontrees` .

source

For a tensor `t` with `FusionStyle(sectortype(t)) isa UniqueFusion` , fusion trees are completely determined by the outcoming sectors, and the data can be accessed in a more straightforward way:

`Base.getindex` – Method.



```
Base.getindex(t::AbstractTensorMap, sectors::Tuple{Vararg{Sector}})
t[sectors]
Base.getindex(t::AbstractTensorMap, f₁::FusionTree, f₂::FusionTree)
t[f₁, f₂]
```

Return a view into the data of `t` corresponding to the splitting - fusion tree pair `(f₁, f₂)` . In particular, this is an `AbstractArray{T}` with `T = scalartype(t)` , of size `(dims(codomain(t), f₁.uncoupled)..., dims(codomain(t), f₂.uncoupled)...)` .

Whenever `FusionStyle(sectortype(t)) isa UniqueFusion` , it is also possible to provide only the external `sectors` , in which case the fusion tree pair will be constructed automatically.

Warning

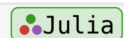
Contrary to Julia's array types, the default behavior is to return a view into the tensor data. As a result, modifying the view will modify the data in the tensor.

See also `subblock` , `subblocks` and `fusiontrees` .

source

For tensor `t` with `sectortype(t) == Trivial` , the data can be accessed and manipulated directly as multidimensional arrays:

`Base.getindex` – Method.



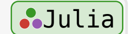
```
Base.getindex(t::AbstractTensorMap)
t[]
```

Return a view into the data of `t` as a `StridedViews.StridedView` of size `dims(t)` .

source

`Base.getindex` – Method.

```
Base.getindex(t::AbstractTensorMap, indices::Vararg{Int})
t[indices]
```

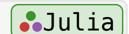


Return a view into the data slice of `t` corresponding to `indices` , by slicing the `StridedViews.StridedView` into the full data array.

source

`Base.setindex!` – Method.

```
Base.setindex!(t::AbstractTensorMap, v, indices::Vararg{Int})
t[indices] = v
```



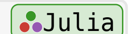
Assigns `v` to the data slice of `t` corresponding to `indices` .

source

The tensor data can also be filled with random numbers via

`Random.rand!` – Function.

```
rand!([rng=default_rng()], t::AbstractTensorMap) -> t
```



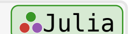
Fill the tensor `t` with entries generated by `rand!` .

See also `Random.rand` .

source

`Random.randn!` – Function.

```
randn!([rng=default_rng()], t::AbstractTensorMap) -> t
```



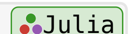
Fill the tensor `t` with entries generated by `randn!` .

See also `Random.randn` .

source

`Random.randexp!` – Function.

```
randexp!([rng=default_rng()], t::AbstractTensorMap) -> t
```



Fill the tensor `t` with entries generated by `randexp!` .

See also `Random.randexp` .

source

15.4 `AbstractTensorMap` operations

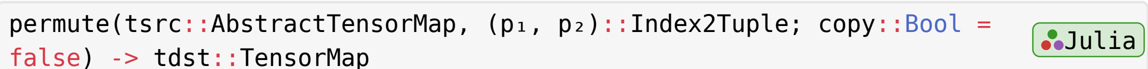
The operations that can be performed on an `AbstractTensorMap` can be organized into the following categories:

- *vector operations*: these do not change the `space` or index structure of a tensor and can be straightforwardly implemented on on the full data. All the methods described in VectorInterface.jl are supported. For compatibility reasons, we also provide implementations for equivalent methods from LinearAlgebra.jl, such as `axpy!` , `axpby!` .
- *index manipulations*: these change (permute) the index structure of a tensor, which affects the data in a way that is fully determined by the categorical data of the `sectortype` of the tensor .
- *(planar) contractions* and *(planar) traces* (i.e., contractions with identity tensors). Tensor contractions correspond to a combination of some index manipulations followed by a composition or multiplication of the tensors in their role as linear maps. Tensor contractions are however of such importance and frequency that they require a dedicated implementation.
- *tensor factorizations*, which relies on their identification of tensors with linear maps between tensor spaces. The factorizations are applied as ordinary matrix factorizations to the matrix blocks associated with the coupled charges.

Index manipulations

A general index manipulation of a `TensorMap` object can be built up by considering some transformation of the fusion trees, along with a permutation of the stored data. They come in three flavours, which are either of the type `transform(!)` which are exported, or of the type `add_transform!` , for additional expert-mode options that allows for addition and scaling, as well as the selection of a custom backend.

`TensorKit.permute` – Method.



```julia
permute(tsrc::AbstractTensorMap, (p₁, p₂)::Index2Tuple; copy::Bool =
false) -> tdst::TensorMap
```

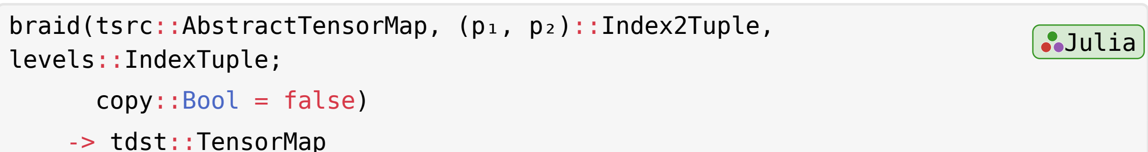
Return tensor `tdst` obtained by permuting the indices of `tsrc` . The codomain and domain of `tdst` correspond to the indices in `p₁` and `p₂` of `tsrc` respectively.

If `copy = false` , `tdst` might share data with `tsrc` whenever possible. Otherwise, a copy is always made.

To permute into an existing destination, see permute! and `add_permute!`

source

`TensorKit.braid` – Method.



```julia
braid(tsrc::AbstractTensorMap, (p₁, p₂)::Index2Tuple,
levels::IndexTuple;
      copy::Bool = false)
    -> tdst::TensorMap
```

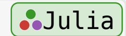
Return tensor `tdst` obtained by braiding the indices of `tsrc` . The codomain and domain of `tdst` correspond to the indices in `p₁` and `p₂` of `tsrc` respectively. Here, `levels` is a tuple of length `numind(tsrc)` that assigns a level or height to the indices of `tsrc` , which determines whether they will braid over or under any other index with which they have to change places.

If `copy=false` , `tdst` might share data with `tsrc` whenever possible. Otherwise, a copy is always made.

To braid into an existing destination, see braid! and `add_braid!`

source

`Base.transpose` – Method.



```julia
transpose(tsrc::AbstractTensorMap, (p₁, p₂)::Index2Tuple;
          copy::Bool=false)
    -> tdst::TensorMap
```

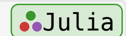
Return tensor `tdst` obtained by transposing the indices of `tsrc` . The codomain and domain of `tdst` correspond to the indices in `p₁` and `p₂` of `tsrc` respectively. The new index positions should be attainable without any indices crossing each other, i.e., the permutation `(p₁..., reverse(p₂)...)` should constitute a cyclic permutation of `(codomainind(tsrc)..., reverse(domainind(tsrc))...)` .

If `copy=false` , `tdst` might share data with `tsrc` whenever possible. Otherwise, a copy is always made.

To permute into an existing destination, see permute! and `add_permute!`

source

`TensorKit.repartition` – Method.



```julia
repartition(
    tsrc::AbstractTensorMap{T, S}, N₁::Int, N₂::Int; copy::Bool=false
) where {T, S} -> tdst::AbstractTensorMap{T, S, N₁, N₂}
```

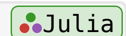
Return tensor `tdst` obtained by repartitioning the indices of `t` . The codomain and domain of `tdst` correspond to the first `N₁` and last `N₂` spaces of `t` , respectively.

If `copy=false` , `tdst` might share data with `tsrc` whenever possible. Otherwise, a copy is always made.

To repartition into an existing destination, see repartition!.

source

`TensorKit.flip` – Method.



```julia
flip(t::AbstractTensorMap, I) -> t′::AbstractTensorMap
```

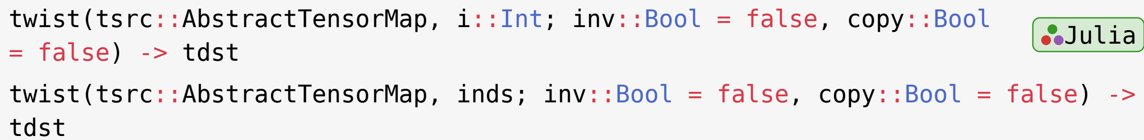
Return a new tensor that is isomorphic to `t` but where the arrows on the indices `i` that satisfy `i ∈ I` are flipped, i.e. `space(t′, i) = flip(space(t, i))` .

Note

The isomorphism that `flip` applies to each of the indices `i ∈ I` is such that flipping two indices that are afterwards contracted within an `@tensor` contraction will yield the same result as without flipping those indices first. However, `flip` is not involutory, i.e. `flip(flip(t, I), I) != t` in general. To obtain the original tensor, one can use the `inv` keyword, i.e. it holds that `flip(flip(t, I), I; inv=true) == t` .

source

`TensorKitSectors.twist` – Method.



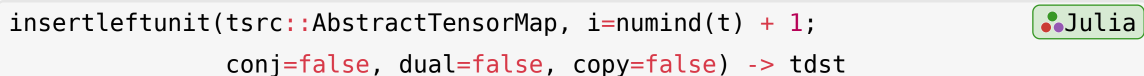
```julia
twist(tsrc::AbstractTensorMap, i::Int; inv::Bool = false, copy::Bool = false) -> tdst
twist(tsrc::AbstractTensorMap, inds; inv::Bool = false, copy::Bool = false) -> tdst
```

Apply a twist to the `i` th index of `tsrc` and return the result as a new tensor. If `inv = true` , use the inverse twist. If `copy = false` , `tdst` might share data with `tsrc` whenever possible. Otherwise, a copy is always made.

See `twist!` for storing the result in place.

source

`TensorKit.insertleftunit` – Method.



```julia
insertleftunit(tsrc::AbstractTensorMap, i=numind(t) + 1;
               conj=false, dual=false, copy=false) -> tdst
```

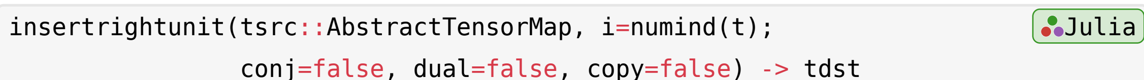
Insert a trivial vector space, isomorphic to the underlying field, at position `i` , which can be specified as an `Int` or as `Val(i)` for improved type stability. More specifically, adds a left monoidal unit or its dual.

If `copy=false` , `tdst` might share data with `tsrc` whenever possible. Otherwise, a copy is always made.

See also `insertrightunit` , `removeunit` .

source

`TensorKit.insertrightunit` – Method.



```julia
insertrightunit(tsrc::AbstractTensorMap, i=numind(t);
                conj=false, dual=false, copy=false) -> tdst
```

Insert a trivial vector space, isomorphic to the underlying field, after position `i` , which can be specified as an `Int` or as `Val(i)` for improved type stability. More specifically, adds a right monoidal unit or its dual.

If `copy=false` , `tdst` might share data with `tsrc` whenever possible. Otherwise, a copy is always made.

See also `insertleftunit` , `removeunit` .

source

`TensorKit.removeunit` – Method.

```julia
removeunit(tsrc::AbstractTensorMap, i; copy=false) -> tdst
```

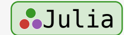
This removes a trivial tensor product factor at position `1 ≤ i ≤ N` , where `i` can be specified as an `Int` or as `Val(i)` for improved type stability. For this to work, that factor has to be isomorphic to the field of scalars.

If `copy=false` , `tdst` might share data with `tsrc` whenever possible. Otherwise, a copy is always made.

This operation undoes the work of `insertleftunit` and `insertrightunit` .

source

`Base.permute!` – Method.



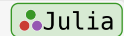
```julia
permute!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap, (p₁,
p₂)::Index2Tuple)
    -> tdst
```

Write into `tdst` the result of permuting the indices of `tsrc` . The codomain and domain of `tdst` correspond to the indices in `p₁` and `p₂` of `tsrc` respectively.

See `permute` for creating a new tensor and `add_permute!` for a more general version.

source

`TensorKit.braid!` – Function.



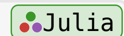
```julia
braid!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap,
       (p₁, p₂)::Index2Tuple, levels::Tuple)
    -> tdst
```

Write into `tdst` the result of braiding the indices of `tsrc` . The codomain and domain of `tdst` correspond to the indices in `p₁` and `p₂` of `tsrc` respectively. Here, `levels` is a tuple of length `numind(tsrc)` that assigns a level or height to the indices of `tsrc` , which determines whether they will braid over or under any other index with which they have to change places.

See `braid` for creating a new tensor and `add_braid!` for a more general version.

source

`LinearAlgebra.transpose!` – Function.



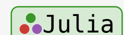
```julia
transpose!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap,
           (p₁, p₂)::Index2Tuple)
    -> tdst
```

Write into `tdst` the result of transposing the indices of `tsrc` . The codomain and domain of `tdst` correspond to the indices in `p₁` and `p₂` of `tsrc` respectively. The new index positions should be attainable without any indices crossing each other, i.e., the permutation `(p₁..., reverse(p₂)...)` should constitute a cyclic permutation of `(codomainind(tsrc)..., reverse(domainind(tsrc))...)` .

See `transpose` for creating a new tensor and `add_transpose!` for a more general version.

source

`TensorKit.repartition!` – Function.



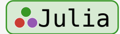
```julia
repartition!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap) ->
tdst
```

Write into `tdst` the result of repartitioning the indices of `tsrc` . This is just a special case of a transposition that only changes the number of in- and outgoing indices.

See `repartition` for creating a new tensor.

source

`TensorKit.twist!` – Function.



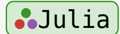
```
twist!(t::AbstractTensorMap, i::Int; inv::Bool=false) -> t
twist!(t::AbstractTensorMap, inds; inv::Bool=false) -> t
```

Apply a twist to the `i` th index of `t` , or all indices in `inds` , storing the result in `t` . If `inv=true` , use the inverse twist.

See `twist` for creating a new tensor.

source

`TensorKit.add_permute!` – Function.



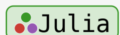
```
add_permute!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap, (p₁,
p₂)::Index2Tuple,
                α::Number, β::Number, backend::AbstractBackend...)
```

Return the updated `tdst` , which is the result of adding `α * tsrc` to `tdst` after permuting the indices of `tsrc` according to `(p₁, p₂)` .

See also `permute` , `permute!` , `add_braid!` , `add_transpose!` .

source

`TensorKit.add_braid!` – Function.



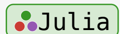
```
add_braid!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap, (p₁,
p₂)::Index2Tuple,
              levels::IndexTuple, α::Number, β::Number,
              backend::AbstractBackend...)
```

Return the updated `tdst` , which is the result of adding `α * tsrc` to `tdst` after braiding the indices of `tsrc` according to `(p₁, p₂)` and `levels` .

See also `braid` , `braid!` , `add_permute!` , `add_transpose!` .

source

`TensorKit.add_transpose!` – Function.



```
add_transpose!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap, (p₁,
p₂)::Index2Tuple,
                α::Number, β::Number, backend::AbstractBackend...)
```

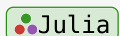
Return the updated `tdst` , which is the result of adding `α * tsrc` to `tdst` after transposing the indices of `tsrc` according to `(p₁, p₂)` .

See also `transpose` , `transpose!` , `add_permute!` , `add_braid!` .

source

Tensor map composition, traces, contractions and tensor products

`TensorKit.compose` – Method.

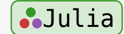


```
compose(t1::AbstractTensorMap, t2::AbstractTensorMap) ->
AbstractTensorMap
```

Return the `AbstractTensorMap` that implements the composition of the two tensor maps `t1` and `t2`.

source

`TensorKit.trace_permute!` – Function.

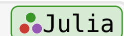


```
trace_permute!(tdst::AbstractTensorMap, tsrc::AbstractTensorMap,
               (p₁, p₂)::Index2Tuple, (q₁, q₂)::Index2Tuple,
               α::Number, β::Number, backend = TO.DefaultBackend())
```

Return the updated `tdst`, which is the result of adding `α * tsrc` to `tdst` after permuting the indices of `tsrc` according to `(p₁, p₂)` and furthermore tracing the indices in `q₁` and `q₂`.

source

`TensorKit.contract!` – Function.

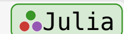


```
contract!(C::AbstractTensorMap,
          A::AbstractTensorMap, (oindA, cindA)::Index2Tuple,
          B::AbstractTensorMap, (cindB, oindB)::Index2Tuple,
          (p₁, p₂)::Index2Tuple,
          α::Number, β::Number,
          backend, allocator)
```

Return the updated `C`, which is the result of adding `α * A * B` to `C` after permuting the indices of `A` and `B` according to `(oindA, cindA)` and `(cindB, oindB)` respectively.

source

`TensorKitSectors.:⊗` – Method.



```
⊗(t1::AbstractTensorMap, t2::AbstractTensorMap, ...) -> TensorMap
otimes(t1::AbstractTensorMap, t2::AbstractTensorMap, ...) -> TensorMap
```

Compute the tensor product between two `AbstractTensorMap` instances, which results in a new `TensorMap` instance whose codomain is `codomain(t1) ⊗ codomain(t2)` and whose domain is `domain(t1) ⊗ domain(t2)`.

source

15.5 `TensorMap` factorizations

The factorization methods are powered by MatrixAlgebraKit.jl and all follow the same strategy. The idea is that the `TensorMap` is interpreted as a linear map based on the current partition of indices between `domain` and `codomain`, and then the entire range of MatrixAlgebraKit functions can be called. Factorizing a tensor according to a different partition of the indices is possible by prepending the factorization step with an explicit call to `permute` or `transpose`.

For the full list of factorizations, see Decompositions.

Additionally, it is possible to obtain truncated versions of some of these factorizations through the `MatrixAlgebraKit.TruncationStrategy` objects.

The exact truncation strategy can be controlled through the strategies defined in Truncations, but for `TensorMap` s there is also the special-purpose scheme:

`TensorKit.Factorizations.truncspace` – Function.

```julia
truncspace(space::ElementarySpace; by=abs, rev::Bool=true)
```

Truncation strategy to keep the first values for each sector when sorted according to `by` and `rev` , such that the resulting vector space is no greater than `V` .

source

Part IV

Index

Chapter 16

Index

- TensorKit.Rep
- TensorKit.Vect
- TensorKitSectors.GroupElement
- TensorKitSectors.Irrep
- TensorKit.AbstractTensor
- TensorKit.AbstractTensorMap
- TensorKit.AdjointTensorMap
- TensorKit.BraidingTensor
- TensorKit.CartesianSpace
- TensorKit.ComplexSpace
- TensorKit.CompositeSpace
- TensorKit.DiagonalTensorMap
- TensorKit.ElementarySpace
- TensorKit.Field
- TensorKit.FusionTree
- TensorKit.GeneralSpace
- TensorKit.GradedSpace
- TensorKit.HomSpace
- TensorKit.InnerProductStyle
- TensorKit.ProductSpace
- TensorKit.Tensor
- TensorKit.TensorMap
- TensorKit.TensorMap
- TensorKit.TensorMap
- TensorKit.TensorMap
- TensorKit.VectorSpace
- TensorKitSectors.AbelianGroup
- TensorKitSectors.AbstractGroupElement
- TensorKitSectors.AbstractIrrep
- TensorKitSectors.BraidingStyle
- TensorKitSectors.CU1Irrep
- TensorKitSectors.CU$_1$
- TensorKitSectors.Cyclic
- TensorKitSectors.DNIrrep

- TensorKitSectors.Dihedral
- TensorKitSectors.FermionNumber
- TensorKitSectors.FermionParity
- TensorKitSectors.FermionSpin
- TensorKitSectors.FibonacciAnyon
- TensorKitSectors.FusionStyle
- TensorKitSectors.Group
- TensorKitSectors.IsingAnyon
- TensorKitSectors.IsingBimodule
- TensorKitSectors.PlanarTrivial
- TensorKitSectors.ProductGroup
- TensorKitSectors.ProductSector
- TensorKitSectors.SU
- TensorKitSectors.SU2Irrep
- TensorKitSectors.Sector
- TensorKitSectors.SectorProductIterator
- TensorKitSectors.SectorValues
- TensorKitSectors.TimeReversed
- TensorKitSectors.Trivial
- TensorKitSectors.U1Irrep
- TensorKitSectors.UnitStyle
- TensorKitSectors.U₁
- TensorKitSectors.ZNElement
- TensorKitSectors.ZNIrrep
- Base.conj
- Base.eltype
- Base.getindex
- Base.getindex
- Base.getindex
- Base.getindex
- Base.isreal
- Base.one
- Base.ones
- Base.permute!
- Base.rand
- Base.randn
- Base.setindex!
- Base.setindex!

- Base.similar
- Base.transpose
- Base.transpose
- Base.zeros
- LinearAlgebra.transpose!
- Random.rand!
- Random.randexp
- Random.randexp!
- Random.randn!
- TensorKit.:⊕
- TensorKit.:⊖
- TensorKit.Factorizations.truncspace
- TensorKit.add_braid!
- TensorKit.add_permute!
- TensorKit.add_transpose!
- TensorKit.allind
- TensorKit.artin_braid
- TensorKit.bendleft
- TensorKit.bendright
- TensorKit.block
- TensorKit.blockdim
- TensorKit.blocks
- TensorKit.blocksectors
- TensorKit.blocksectors
- TensorKit.blocksectors
- TensorKit.blocktype
- TensorKit.braid
- TensorKit.braid
- TensorKit.braid
- TensorKit.braid!
- TensorKit.codomain
- TensorKit.codomainind
- TensorKit.compose
- TensorKit.compose
- TensorKit.contract!
- TensorKit.dims
- TensorKit.domain
- TensorKit.domainind

- TensorKit.elementary_trace
- TensorKit.field
- TensorKit.field
- TensorKit.flip
- TensorKit.flip
- TensorKit.flip
- TensorKit.foldleft
- TensorKit.foldright
- TensorKit.fuse
- TensorKit.fusiontrees
- TensorKit.fusiontrees
- TensorKit.fusiontrees
- TensorKit.hasblock
- TensorKit.hasblock
- TensorKit.hassector
- TensorKit.id
- TensorKit.infimum
- TensorKit.insertat
- TensorKit.insertleftunit
- TensorKit.insertleftunit
- TensorKit.insertleftunit
- TensorKit.insertrightunit
- TensorKit.insertrightunit
- TensorKit.insertrightunit
- TensorKit.isconj
- TensorKit.isdual
- TensorKit.isepimorphic
- TensorKit.isisomorphic
- TensorKit.ismonomorphic
- TensorKit.isometry
- TensorKit.isomorphism
- TensorKit.join
- TensorKit.merge
- TensorKit.numin
- TensorKit.numind
- TensorKit.numout
- TensorKit.permute
- TensorKit.permute

- TensorKit.permute
- TensorKit.permute
- TensorKit.planar_trace
- TensorKit.reduceddim
- TensorKit.removeunit
- TensorKit.removeunit
- TensorKit.removeunit
- TensorKit.repartition
- TensorKit.repartition
- TensorKit.repartition!
- TensorKit.sectors
- TensorKit.sectortype
- TensorKit.sectortype
- TensorKit.select
- TensorKit.space
- TensorKit.space
- TensorKit.spacetype
- TensorKit.split
- TensorKit.storagetype
- TensorKit.subblock
- TensorKit.subblocks
- TensorKit.supremum
- TensorKit.trace_permute!
- TensorKit.twist!
- TensorKit.unitary
- TensorKit.unitspace
- TensorKit.zerospace
- TensorKitSectors.:×
- TensorKitSectors.:⊗
- TensorKitSectors.:⊗
- TensorKitSectors.:⊗
- TensorKitSectors.:⊠
- TensorKitSectors.Bsymbol
- TensorKitSectors.Fsymbol
- TensorKitSectors.Nsymbol
- TensorKitSectors.Rsymbol
- TensorKitSectors.allunits
- TensorKitSectors.charge

- TensorKitSectors.deligneproduct
- TensorKitSectors.dim
- TensorKitSectors.dim
- TensorKitSectors.dim
- TensorKitSectors.dim
- TensorKitSectors.dim
- TensorKitSectors.dim
- TensorKitSectors.dual
- TensorKitSectors.dual
- TensorKitSectors.findindex
- TensorKitSectors.frobenius_schur_indicator
- TensorKitSectors.frobenius_schur_phase
- TensorKitSectors.isunit
- TensorKitSectors.leftunit
- TensorKitSectors.modulus
- TensorKitSectors.precompile_sector
- TensorKitSectors.rightunit
- TensorKitSectors.sectorscalartype
- TensorKitSectors.twist
- TensorKitSectors.twist
- TensorKitSectors.type_repr
- TensorKitSectors.unit

Part V

Appendix

Chapter 17

A symmetric tensor deep dive: constructing your first tensor map

In this tutorial, we will demonstrate how to construct specific `TensorMap` s which are relevant to some common physical systems, with an increasing degree of complexity. We will assume the reader is somewhat familiar with the notion of a *tensor map* and has a rough idea of what it means for a tensor map to be *symmetric*. In going through these examples we aim to provide a relatively gentle introduction to the meaning of symmetry sectors and vector spaces within the context of TensorKit.jl, how to initialize a `TensorMap` over a given vector space and finally how to manually set the data of a symmetric `TensorMap` . We will keep our discussion as intuitive and simple as possible, only adding as many technical details as strictly necessary to understand each example. When considering a different physical system of interest, you should then be able to adapt these recipes and the intuition behind them to your specific problem at hand.

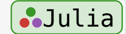
Note

Many of these examples are readily implemented in the TensorKitTensors.jl package, in which case we basically provide a narrated walk-through of the corresponding code.

Contents of the tutorial

- Level 0: The transverse-field Ising model
- Level 1: The $\mathbb{Z}_2$-symmetric Ising model
  - The irrep basis and block sparsity
  - Fusion trees and how to use them
  - Constructing a $\mathbb{Z}_2$-symmetric `TensorMap`
- Level 2: The U_1 Bose-Hubbard model
  - Directly constructing the Hamiltonian terms
  - Creation and annihilation operators as symmetric tensors
- Level 3: Fermions and the Kitaev model
  - Fermion parity symmetry
  - Constructing the Hamiltonian
- Level 4: Non-Abelian symmetries and the quantum Heisenberg model
  - Block sparsity revisited: the Wigner-Eckart theorem
  - The 'generic' approach to the spin-1 Heisenberg model: Wigner-Eckart in action
  - An 'elegant' approach to the Heisenberg model
  - SU_N generalization
- Level 5: Anyonic Symmetries and the Golden Chain

Setup



```
using LinearAlgebra
using TensorKit
using WignerSymbols
using SUNRepresentations
using Test # for showcase testing
```

17.1 Level 0: The transverse-field Ising model

As the most basic example, we consider the 1-dimensional transverse-field Ising model, whose Hamiltonian is given by

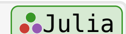
$$H = -J\left(\sum_{\langle i,j\rangle} Z_i Z_j + g\sum_i X_i\right).$$

Here, X_i and Z_i are the Pauli operators acting on site i, and the first sum runs over pairs of nearest neighbors $\langle i,j\rangle$. This model has a global $\mathbb{Z}_2$ symmetry, as it is invariant under the transformation $UHU^\dagger = H$ where the symmetry transformation U is given by a global spin flip,

$$U = \prod_i X_i.$$

We will circle back to the implications of this symmetry later.

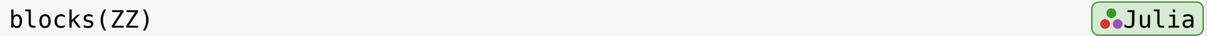
As a warmup, we implement the Hamiltonian \eqref{eq:isingham} in the standard way by encoding the matrix elements of the single-site operators X and Z into an array of complex numbers, and then combine them in a suitable way to get the Hamiltonian terms. Instead of using plain Julia arrays, we use a representation in terms of `TensorMap` s over complex vector spaces. These are essentially just wrappers around base arrays at this point, but their construction requires some consideration of the notion of *spaces*, which generalize the notion of `size` for arrays. Each of the operators X and Z acts on a local two-dimensional complex vector space. In the context of TensorKit.jl, such a space can be represented as `ComplexSpace(2)` , or using the convenient shorthand `ℂ^2` . A single-site Pauli operator maps from a domain physical space to a codomain physical space, and can therefore be represented as instances of a `TensorMap(..., ℂ^2 ← ℂ^2)` . The corresponding data can then be filled in by hand according to the familiar Pauli matrices in the following way:



```
# initialize numerical data for Pauli matrices
x_mat = ComplexF64[0 1; 1 0]
z_mat = ComplexF64[1 0; 0 -1]

# construct physical Hilbert space
V = ℂ^2

# construct the physical operators as TensorMaps
X = TensorMap(x_mat, V ← V)
Z = TensorMap(z_mat, V ← V)

# combine single-site operators into two-site operator
ZZ = Z ⊗ Z
```

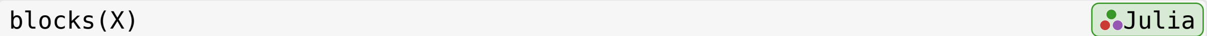
```
2×2←2×2 TensorMap{ComplexF64, ComplexSpace, 2, 2,
Vector{ComplexF64}}:
 codomain: (ℂ^2 ⊗ ℂ^2)
 domain: (ℂ^2 ⊗ ℂ^2)
 blocks:
 * Trivial() => 4×4 reshape(view(::Vector{ComplexF64}, 1:16), 4, 4) with eltype
 ComplexF64:
 1.0+0.0im   0.0+0.0im   0.0+0.0im  0.0+0.0im
 0.0+0.0im  -1.0+0.0im   0.0+0.0im  0.0+0.0im
 0.0+0.0im   0.0+0.0im  -1.0+0.0im  0.0+0.0im
 0.0+0.0im   0.0+0.0im   0.0+0.0im  1.0+0.0im
```

We can easily verify that our operators have the desired form by checking their data in the computational basis. We can print this data by calling the `blocks` method (we'll explain exactly what these *blocks* are further down):



```julia
blocks(ZZ)
```

```
blocks(::TensorMap{ComplexF64, ComplexSpace, 2, 2,
Vector{ComplexF64}}):
 * Trivial() => 4×4 reshape(view(::Vector{ComplexF64}, 1:16), 4, 4) with eltype
 ComplexF64:
 1.0+0.0im   0.0+0.0im   0.0+0.0im  0.0+0.0im
 0.0+0.0im  -1.0+0.0im   0.0+0.0im  0.0+0.0im
 0.0+0.0im   0.0+0.0im  -1.0+0.0im  0.0+0.0im
 0.0+0.0im   0.0+0.0im   0.0+0.0im  1.0+0.0im
```



```julia
blocks(X)
```

```
blocks(::TensorMap{ComplexF64, ComplexSpace, 1, 1,
Vector{ComplexF64}}):
 * Trivial() => 2×2 reshape(view(::Vector{ComplexF64}, 1:4), 2, 2) with eltype
 ComplexF64:
 0.0+0.0im  1.0+0.0im
 1.0+0.0im  0.0+0.0im
```

17.2 Level 1: The $\mathbb{Z}_2$-symmetric Ising model

The irrep basis and block sparsity

Let us now return to the global $\mathbb{Z}_2$ invariance of the Hamiltonian \eqref{eq:isingham}, and consider what this implies for its local terms ZZ and X. Representing these operators as `TensorMap` s, the invariance of H under a global $\mathbb{Z}_2$ transformation implies the following identities for the local tensors:

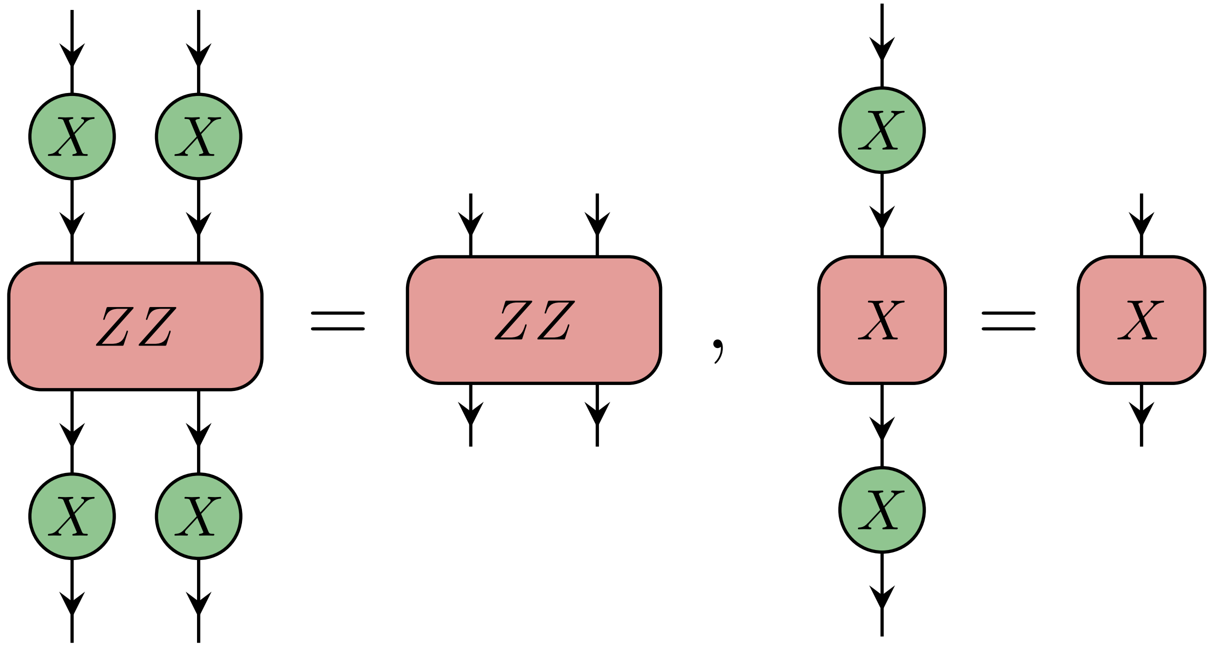


Figure 17.1: ZZX_symm

These identitities precisely mean that these local tensors transform trivially under a tensor product representation of $\mathbb{Z}_2$. This implies that, recalling the introduction on symmetries, in an appropriate basis for the local physical vector space, our local tensors would become block-diagonal where each so-called *matrix block* is labeled by a $\mathbb{Z}_2$ irrep. The appropriate local basis transformation is precisely the one that brings the local representation X into block-diagonal form. Clearly, this transformation is nothing more than the Hadamard transformation which maps the computational basis of Z eigenstates $\{|\uparrow\rangle, |\downarrow\rangle\}$ to that of the X eigenstates $\{|+\rangle, |-\rangle\}$ defined as $|+\rangle = \frac{|\uparrow\rangle + |\downarrow\rangle}{\sqrt{2}}$ and $|-\rangle = \frac{|\uparrow\rangle - |\downarrow\rangle}{\sqrt{2}}$. In the current context, this basis is referred to as the *irrep basis* of $\mathbb{Z}_2$, since each basis state corresponds to a one-dimensional irreducible representation of $\mathbb{Z}_2$. Indeed, the local symmetry transformation X acts trivially on the state $|+\rangle$, corresponding to the *trivial irrep*, and yields a minus sign when acting on $|-\rangle$, corresponding to the *sign irrep*.

Next, let's make the statement that "the matrix blocks of the local tensors are labeled by $\mathbb{Z}_2$ irreps" more concrete. To this end, consider the action of ZZ in the irrep basis, which is given by the four nonzero matrix elements

$$
\begin{aligned}
&ZZ : \mathbb{C}^2 \otimes \mathbb{C}^2 \to \mathbb{C}^2 \otimes \mathbb{C}^2 : \\
&\quad |+\rangle \otimes |+\rangle \mapsto |-\rangle \otimes |-\rangle, \\
&\quad |+\rangle \otimes |-\rangle \mapsto |-\rangle \otimes |+\rangle, \\
&\quad |-\rangle \otimes |+\rangle \mapsto |+\rangle \otimes |-\rangle, \\
&\quad |-\rangle \otimes |-\rangle \mapsto |+\rangle \otimes |+\rangle.
\end{aligned}
$$

We will denote the trivial $\mathbb{Z}_2$ irrep by $'0'$, corresponding to a local $|+\rangle$ state, and the sign irrep by $'1'$, corresponding to a local $|-\rangle$ state. Given this identification, we can naturally associate the tensor product of basis vectors in the irrep basis to the tensor product of the corresponding $\mathbb{Z}_2$ irreps. One of the key questions of the representation theory of groups is how the tensor product of two irreps can be decomposed into a direct sum of irreps. This decomposition is encoded in what are often called the *fusion rules*,

$$
a \otimes b \cong \bigoplus_c N_c^{ab} c,
$$

where N_{ab}^{c} encodes the number of times the irrep c occurs in the tensor product of irreps a and b. These fusion rules are called *Abelian* if the tensor product of any two irreps corresponds to exactly one irrep. We will return to the implications of irreps with *non-Abelian* fusion rules later.

Note

Within TensorKit.jl, the nature of the fusion rules for charges of a given symmetry are represented by the `FusionStyle` of the corresponding `Sector` subtype. What we refer to as "Abelian" fusion rules in this tutorial corresponds to `UniqueFusion <: FusionStyle` . We will also consider examples of two different kinds of non-Abelian" fusion rules, corresponding to `MultipleFusion <: FusionStyle` styles.

For the case of the $\mathbb{Z}_2$ irreps, the fusion rules are Abelian, and are given by addition modulo 2,

$$0 \otimes 0 \cong 0, \quad 0 \otimes 1 \cong 1, \quad 1 \otimes 0 \cong 1, \quad 1 \otimes 1 \cong 0.$$

To see how these fusion rules arise, we can consider the action of the symmetry transformation XX on the possible two-site basis states, each of which corresponds to a tensor product of representations. We can see that XX acts trivially on both $|+\rangle \otimes |+\rangle$ and $|-\rangle \otimes |-\rangle$, meaning these transform under the trivial representation, which gives the first and last entries of the fusion rules. Similarly, XX acts with a minus sign on both $|+\rangle \otimes |-\rangle$ and $|-\rangle \otimes |+\rangle$, meaning these transform under the sign representation, which gives the second and third entries of the fusion rules. Having introduced this notion of 'fusing' irreps, we can now associate a well-defined *coupled irrep* to each of the four two-site basis states, which is given by the tensor product of the two *uncoupled irreps* associated to each individual site. From the matrix elements of ZZ given above, we clearly see that this operator only maps between states in the domain and codomain that have the same coupled irrep. This means that we can associate each of these matrix elements to a so-called *fusion tree* of $\mathbb{Z}_2$ irreps with a corresponding coefficient of 1,

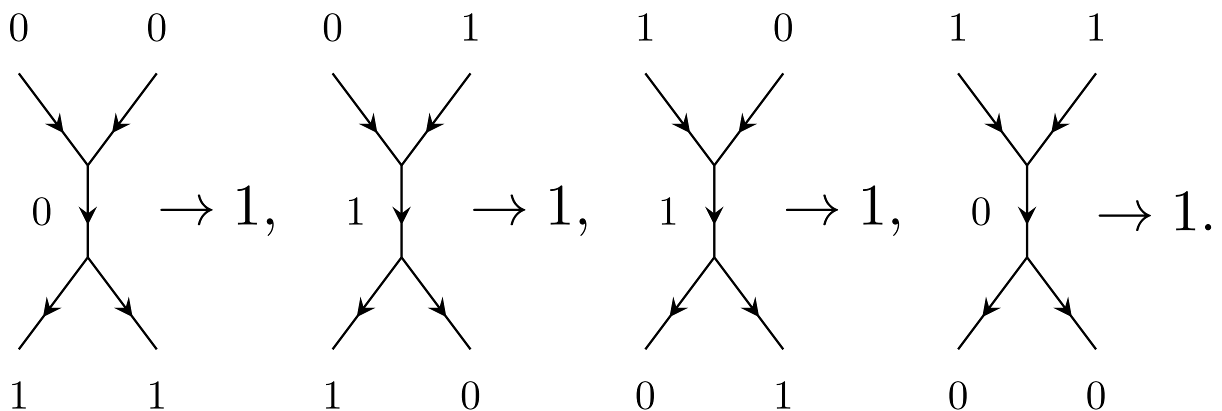


Figure 17.2: Z2_fusiontrees

This diagram should be read from top to bottom, where it represents the fusion of the two uncoupled irreps in the domain to the coupled irrep on the middle line, and the splitting of this coupled irrep to the uncoupled irreps in the codomain. From this our previous statement becomes very clear: the ZZ operator indeed consists of two distinct two-dimensional matrix blocks, each of which are labeled by the value of the *coupled irrep* on the middle line of each fusion tree. The first block corresponds to the even coupled irrep '0', and acts within the two-dimensional subspace spanned by $\{|+,+\rangle, |-,-\rangle\}$, while the second block corresponds to the odd coupled irrep '1', and acts within the two-dimensional subspace spanned by $\{|+,-\rangle, |-,+\rangle\}$. In TensorKit.jl, this block-diagonal structure of a symmetric tensor is explicitly encoded into its representation as a `TensorMap` , where only the matrix blocks

corresponding to each coupled irrep are stored. These matrix blocks associated to each coupled irrep are precisely what is accessed by the `blocks` method we have already used above.

For our current purposes however, *we never really need to explicitly consider these matrix blocks.* Indeed, when constructing a `TensorMap` it is sufficient to set its data by manually assigning a matrix element to each fusion tree of the form above labeled by a given tensor product of irreps. This matrix element is then automatically inserted into the appropriate matrix block. So, for the purpose of this tutorial **we will interpret a symmetric `TensorMap` simply as a list of fusion trees, to each of which corresponds a certain reduced tensor element**. In TensorKit.jl, these reduced tensor elements corresponding to the fusion trees of a `TensorMap` can be accessed through the `subblocks` method.

> **Note**
>
> In general, such a reduced tensor element is not necessarily a scalar, but rather an array whose size is determined by the degeneracy of the irreps in the codomain and domain of the fusion tree. For this reason, a reduced tensor element associated to a given fusion tree is also referred to as a *subblock*. In the following we will always use terms *reduced tensor element* or *subblock* for the reduced tensor elements, to make it clear that these are distinct from the matrix blocks in the block-diagonal decomposition of the tensor.

Fusion trees and how to use them

This view of the underlying symmetry structure in terms of fusion trees of irreps and corresponding reduced tensor elements is a very convenient way of working with the `TensorMap` type. In fact, this symmetry structure is inherently ingrained in a `TensorMap` , and goes beyond the group-like symmetries we have considered until now. In this more general setting, we will refer to the labels that appear on this fusion trees as *charges* or *sectors*. These can be thought of as generalization of group irreps, and appear in the context of TensorKit.jl as instances of the `Sector` type.

Consider a generic fusion tree of the form

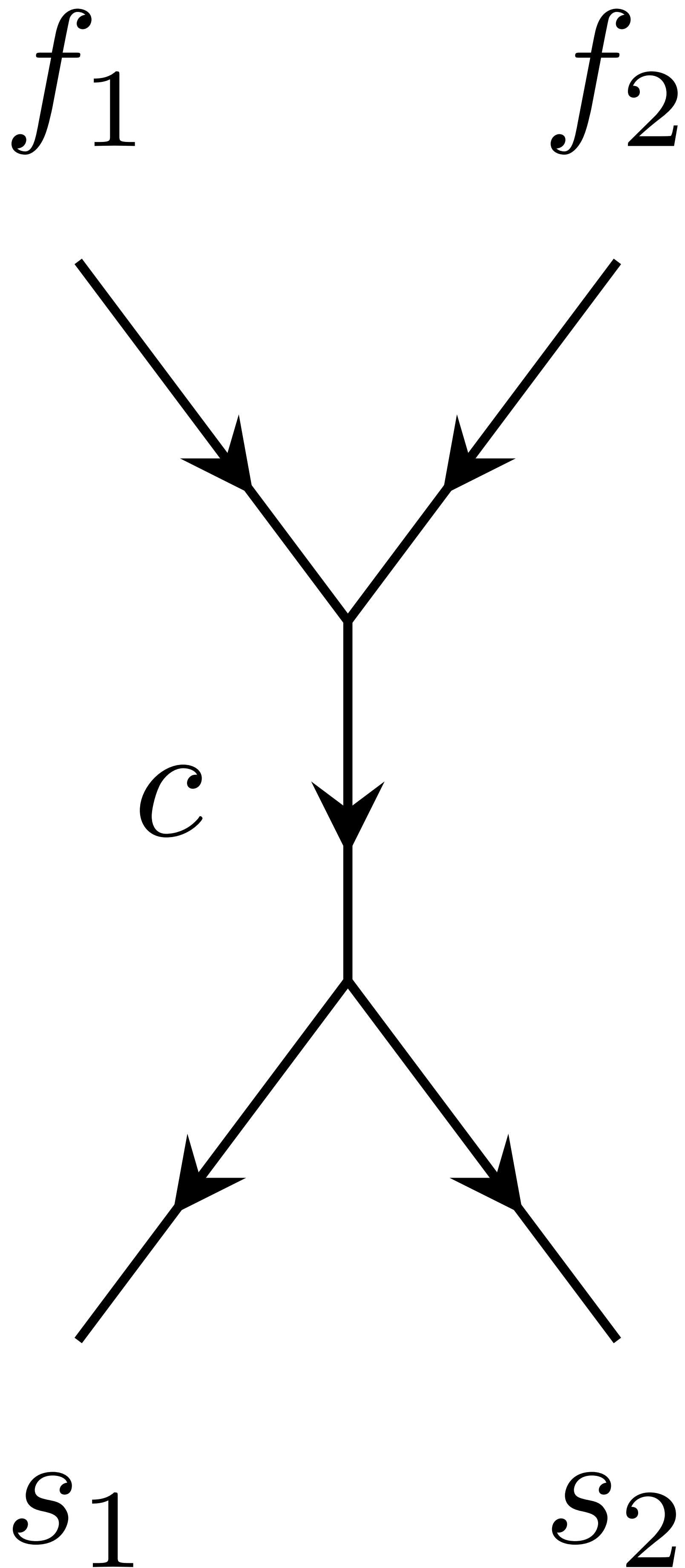


Figure 17.3: fusiontree

which can be used to label a subblock of a `TensorMap` corresponding to a two-site operator. This object should actually be seen as a *pair of fusion trees*. The first member of the pair, related to the codomain of the `TensorMap` , is referred to as the *splitting tree* and encodes how the *coupled charge* c splits into the *uncoupled charges* s_1 and s_2. The second member of the pair, related to the domain of the `TensorMap` , is referred to as the *fusion tree* and encodes how the uncoupled charges f_1 and f_2 fuse to the coupled charge c. Both the splitting and fusion tree can be represented as a `FusionTree` instance. You will find such a `FusionTree` has the following properties encoded into its fields:

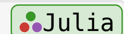
- `uncoupled::NTuple{N,I}` : a list of `N` uncoupled charges of type `I<:Sector`
- `coupled::I` : a single coupled charge of type `I<:Sector`
- `isdual::NTuple{N,Bool}` : a list of booleans indicating whether the corresponding uncoupled charge is dual
- `innerlines::NTuple{M,I}` : a list of inner lines of type `I<:Sector` of length `M = N - 2`
- `vertices::NTuple{L,T}` : list of fusion vertex labels of type `T` and length `L = N - 1`

For our current application only `uncoupled` and `coupled` are relevant, since $\mathbb{Z}_2$ irreps are self-dual and have Abelian fusion rules, so that irreps on the inner lines of a fusion tree are completely determined by the uncoupled irreps. We will come back to these other properties when discussion more involved applications. Given some `TensorMap` , the method `fusiontrees` returns an iterator over all pairs of splitting and fusion trees that label the subblocks of `t` .

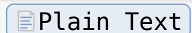
Constructing a $\mathbb{Z}_2$-symmetric `TensorMap`

We can now put this into practice by directly constructing the ZZ operator in the irrep basis as a $\mathbb{Z}_2$ -symmetric `TensorMap` . We will do this in three steps:
1. First we construct the physical space at each site as a $\mathbb{Z}_2$-graded vector space.
2. Then we initialize an empty `TensorMap` with the correct domain and codomain vector spaces built from the previously constructed physical space.
3. And finally we iterate over all splitting and fusion tree pairs and manually fill in the corresponding nonzero subblocks of the operator.

In TensorKit.jl, the representations of $\mathbb{Z}_2$ are represented as instances of the `Z2Irrep <: Sector` type. There are two such instances, corresponding to the trivial irrep `Z2Irrep(0)` and the sign irrep `Z2Irrep(1)` . We can fuse irreps with the `⊗` (`\otimes`) operator, which can for example be used to check their fusion rules,



```julia
for a in values(Z2Irrep), b in values(Z2Irrep)
    println("$a ⊗ $b = $(a ⊗ b)")
end
```



```
Irrep[ℤ₂](0) ⊗ Irrep[ℤ₂](0) = (Irrep[ℤ₂](0),)
Irrep[ℤ₂](0) ⊗ Irrep[ℤ₂](1) = (Irrep[ℤ₂](1),)
Irrep[ℤ₂](1) ⊗ Irrep[ℤ₂](0) = (Irrep[ℤ₂](1),)
Irrep[ℤ₂](1) ⊗ Irrep[ℤ₂](1) = (Irrep[ℤ₂](0),)
```

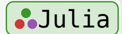
After the basis transform to the irrep basis, we can view the two-dimensional complex physical vector space we started with as being spanned by the trivial and sign irrep of $\mathbb{Z}_2$. In the language of TensorKit.jl, this can be implemented as a `Z2Space` , an alias for a graded vector space `Vect[Z2Irrep]` . Such a graded vector space V is a direct sum of irreducible representation spaces $V^{(a)}$ labeled by the irreps a of the group,

$$V = \bigotimes_a N_a \cdot V^{(a)}.$$

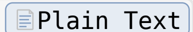
The number of times N_a each irrep a appears in the direct sum is called the *degeneracy* of the irrep. To construct such a graded space, we therefore have to specify which irreps it contains, and indicate the degeneracy of each irrep. Here, our physical vector space contains the trivial irrep `Z2Irrep(0)` with degeneracy 1 and the sign irrep `Z2Irrep(1)` with degeneracy 1. This means this particular graded space has the form

$$V = 1 \cdot V^{(0)} \oplus 1 \cdot V^{(1)},$$

which can be constructed in the following way,

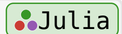


```
V = Z2Space(0 => 1, 1 => 1)
```

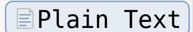


```
Rep[ℤ₂](…) of dim 2:
 0 => 1
 1 => 1
```

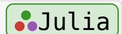
As a consistency check, we can inspect its dimension as well as the degeneracies of the individual irreps:



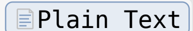
```
dim(V)
```



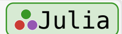
```
2
```



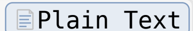
```
dim(V, Z2Irrep(0))
```



```
1
```

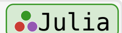


```
dim(V, Z2Irrep(1))
```

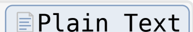


```
1
```

Given this physical space, we can initialize the ZZ operator as an empty `TensorMap` with the appropriate structure.



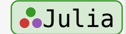
```
ZZ = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
```



```
2×2←2×2 TensorMap{ComplexF64, Rep[ℤ₂], 2, 2, Vector{ComplexF64}}:
 codomain: (Rep[ℤ₂](0 => 1, 1 => 1) ⊗ Rep[ℤ₂](0 => 1, 1 => 1))
 domain: (Rep[ℤ₂](0 => 1, 1 => 1) ⊗ Rep[ℤ₂](0 => 1, 1 => 1))
 blocks:
 * Irrep[ℤ₂](0) => 2×2 reshape(view(::Vector{ComplexF64}, 1:4), 2, 2) with eltype
 ComplexF64:
 0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im

 * Irrep[ℤ₂](1) => 2×2 reshape(view(::Vector{ComplexF64}, 5:8), 2, 2) with eltype
 ComplexF64:
 0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im
```

To assess the underlying structure of a symmetric tensor, it is often useful to inspect its `subblocks` ,

```
subblocks(ZZ)
```

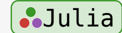


Plain Text

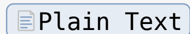
```
subblocks(::TensorMap{ComplexF64, Rep[ℤ₂], 2, 2,
Vector{ComplexF64}}):
 * (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im
```

While all entries are zero, we see that all eight valid fusion trees with two incoming irreps and two outgoing irreps of the type above are listed with their corresponding subblock data. Each of these

subblocks is an array of shape $(1, 1, 1, 1)$ since each irrep occuring in the space V has degeneracy 1. Using the `fusiontrees` method and the fact that we can index a `TensorMap` using a splitting/fusion tree pair, we can now fill in the nonzero subblocks of the operator by observing that the ZZ operator flips the irreps of the uncoupled charges in the domain with respect to the codomain, as shown in the diagrams above. Flipping a given `Z2Irrep` in the codomain can be implemented by fusing them with the sign irrep `Z2Irrep(1)`, giving:



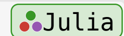
```julia
flip_charge(charge::Z2Irrep) = only(charge ⊗ Z2Irrep(1))
for (s, f) in fusiontrees(ZZ)
    if s.uncoupled == map(flip_charge, f.uncoupled)
        ZZ[s, f] .= 1
    end
end
subblocks(ZZ)
```



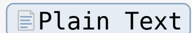
```
subblocks(::TensorMap{ComplexF64, Rep[ℤ₂], 2, 2,
Vector{ComplexF64}}):
 * (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 1.0 + 0.0im

 * (Irrep[ℤ₂](0), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 1.0 + 0.0im

 * (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](1)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 1.0 + 0.0im
```

```
 * (Irrep[ℤ₂](1), Irrep[ℤ₂](0)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 1.0 + 0.0im

 * (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) ← (Irrep[ℤ₂](0), Irrep[ℤ₂](1)) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im
```

Indeed, the resulting `TensorMap` exactly encodes the matrix elements of the ZZ operator shown in the diagrams above. The X operator can be constructed in a similar way. Since it is by definition diagonal in the irrep basis with matrix blocks directly corresponding to the trivial and sign irrep, its construction is particularly simple:



```
X = zeros(ComplexF64, V ← V)
for (s, f) in fusiontrees(X)
    if only(f.uncoupled) == Z2Irrep(0)
        X[s, f] .= 1
    else
        X[s, f] .= -1
    end
end
subblocks(X)
```



```
subblocks(::TensorMap{ComplexF64, Rep[ℤ₂], 1, 1,
Vector{ComplexF64}}):
 * (Irrep[ℤ₂](0),) ← (Irrep[ℤ₂](0),) => 1×1 StridedViews.StridedView{ComplexF64,
 2, Memory{ComplexF64}, typeof(identity)}:
 1.0 + 0.0im

 * (Irrep[ℤ₂](1),) ← (Irrep[ℤ₂](1),) => 1×1 StridedViews.StridedView{ComplexF64,
 2, Memory{ComplexF64}, typeof(identity)}:
 -1.0 + 0.0im
```

Given these local operators, we can use them to construct the full manifestly $\mathbb{Z}_2$-symmetric Hamiltonian.

Note

An important observation is that, when explicitly imposing the $\mathbb{Z}_2$ symmetry, we directly constructed the full ZZ operator as a single symmetric tensor. This in contrast to the case without symmetries, where we constructed a single-site Z operator and then combined them into a two-site operator. Clearly this can no longer be done when imposing $\mathbb{Z}_2$, since a single Z is not invariant under conjugation with the symmetry operator X. One might wonder whether it is still possible to construct a two-site Hamiltonian term by combining local objects. This is possible if one introduces an auxiliary index on the local tensors that carries a non-trivial charge. The intuition behind this will become more clear in the next example.

17.3 Level 2: The U_1 Bose-Hubbard model

For our next example, we consider the Bose-Hubbard model, which describes interacting bosons on a lattice. The Hamiltonian of this model is given by

$$H = -t \sum_{\langle i,j \rangle} \left(a_i^+ a_j^- + a_i^- a_j^+ \right) - \mu \sum_i N_i + \frac{U}{2} \sum_i N_i (N_i - 1).$$

This Hamiltonian is defined on the Fock space associated to a chain of bosons, where the action bosonic creation, annihilation and number operators a^+, a^- and $N = a^+ a^-$ in the local occupation number basis is given by

$$\begin{aligned} a^+ \, |n\rangle &= \sqrt{n+1} \, |n+1\rangle \\ a^- \, |n\rangle &= \sqrt{n} \, |n-1\rangle \\ N \, |n\rangle &= n \, |n\rangle \end{aligned}$$

Their bosonic nature can be summarized by the familiar the commutation relations

$$\begin{aligned} \left[a_i^-, a_j^-\right] &= \left[a_i^+, a_j^+\right] = 0 \\ \left[a_i^-, a_j^+\right] &= \delta_{ij} \\ \left[N, a^+\right] &= a^+ \\ \left[N, a^-\right] &= -a^- \end{aligned}$$

This Hamiltonian is invariant under conjugation by the global particle number operator, $UHU^\dagger = H$, where

$$U = \sum_i N_i$$

This invariance corresponds to a U_1 particle number symmetry, which can again be manifestly imposed when constructing the Hamiltonian terms as `TensorMap` s. From the representation theory of U_1, we know that its irreps are all one-dimensional and can be labeled by integers n where the tensor product of two irreps is corresponds to addition of these labels, giving the Abelian fusion rules

$$n_1 \otimes n_2 \cong (n_1 + n_2).$$

Directly constructing the Hamiltonian terms

We recall from our discussion on the $\mathbb{Z}_2$ symmetric Ising model that, in order to construct the Hamiltonian terms as symmetric tensors, we should work in the irrep basis where the symmetry transformation is block diagonal. In the current case, the symmetry operation is the particle number operator, which is already diagonal in the occupation number basis. Therefore, we don't need an additional local basis transformation this time, and can just observe that each local basis state can be identified with the U_1 irrep associated to the corresponding occupation number.

Following the same approach as before, we first write down the action of the Hamiltonian terms in the irrep basis:

$$a_i^+ a_j^- \; |n_i, n_j\rangle = \sqrt{(n_i+1)n_j} \; |n_i+1, n_j-1\rangle$$
$$a_i^- a_j^+ \; |n_i, n_j\rangle = \sqrt{n_i(n_j+1)} \; |n_i-1, n_j+1\rangle$$
$$N \; |n\rangle = n \; |n\rangle$$

It is then a simple observation that these matrix elements are exactly captured by the following U_1 fusion trees with corresponding subblock values:

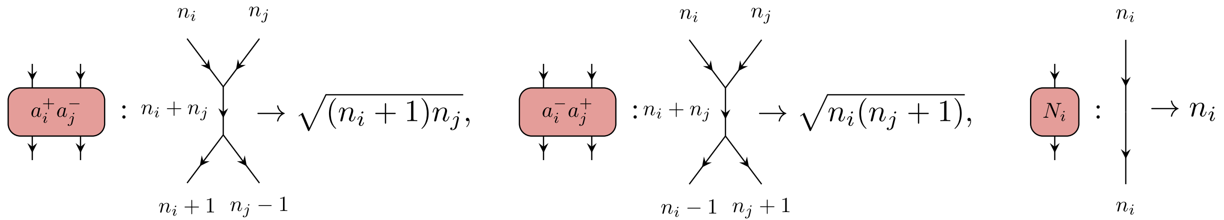


Figure 17.4: U1_fusiontrees

This gives us all the information necessary to construct the corresponding `TensorMap` s. We follow the same steps as outlined in the previous example, starting with the construction of the physical space. This will now be a U_1 graded vector space `U1Space` , where each basis state $|n\rangle$ in the occupation number basis is represented by the corresponding U_1 irrep `U1Irrep(n)` with degeneracy 1. While this physical space is in principle infinite dimensional, we will impose a cutoff in occupation number at a maximum of 5 bosons per site, giving a 6-dimensional vector space:

```julia
cutoff = 5
V = U1Space(n => 1 for n in 0:cutoff)
```

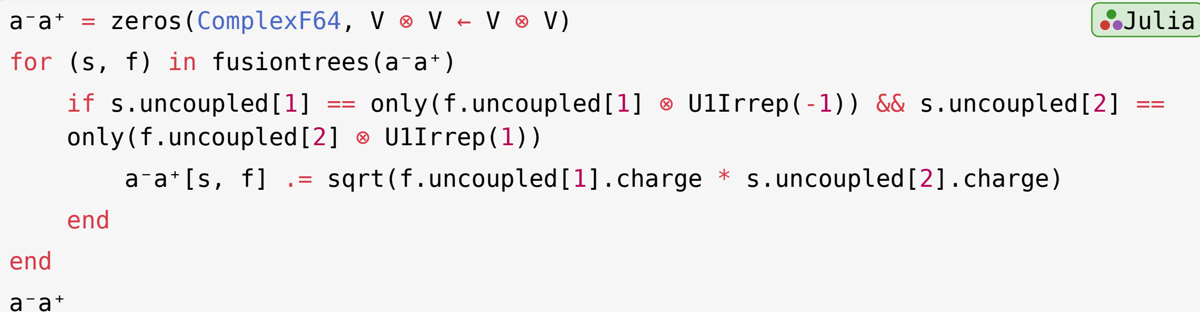
```
Rep[U₁](…) of dim 6:
 0 => 1
 1 => 1
 2 => 1
 3 => 1
 4 => 1
 5 => 1
```

We can now initialize the a^+a^-, a^-a^+ and N operators as empty `TensorMap` s with the correct domain and codomain vector spaces, and fill in the nonzero subblocks associated to the fusion trees shown above. To do this we need access to the integer label of the U_1 irreps in the fusion and splitting trees, which can be accessed through the `charge` field of the `U1Irrep` type.

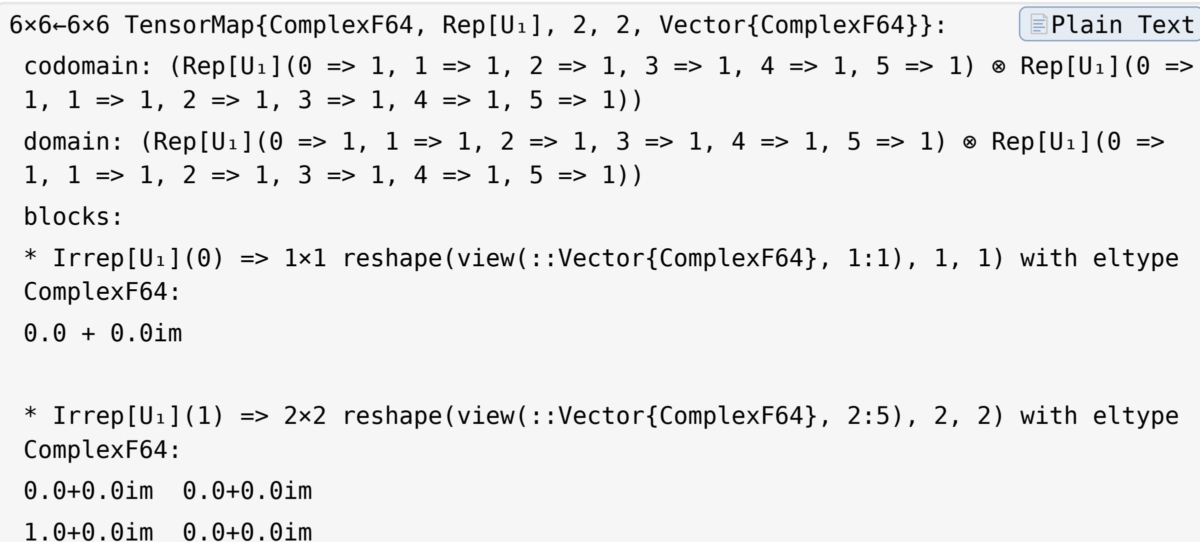
```julia
a⁺a⁻ = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
for (s, f) in fusiontrees(a⁺a⁻)
    if s.uncoupled[1] == only(f.uncoupled[1] ⊗ U1Irrep(1)) && s.uncoupled[2] ==
    only(f.uncoupled[2] ⊗ U1Irrep(-1))
        a⁺a⁻[s, f] .= sqrt(s.uncoupled[1].charge * f.uncoupled[2].charge)
    end
end
a⁺a⁻
```

```
6×6←6×6 TensorMap{ComplexF64, Rep[U₁], 2, 2, Vector{ComplexF64}}:
```

```
codomain: (Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1) ⊗ Rep[U₁](0 =>
1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1))
domain: (Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1) ⊗ Rep[U₁](0 =>
1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1))
blocks:
* Irrep[U₁](0) => 1×1 reshape(view(::Vector{ComplexF64}, 1:1), 1, 1) with eltype
ComplexF64:
0.0 + 0.0im

* Irrep[U₁](1) => 2×2 reshape(view(::Vector{ComplexF64}, 2:5), 2, 2) with eltype
ComplexF64:
0.0+0.0im  1.0+0.0im
0.0+0.0im  0.0+0.0im

* Irrep[U₁](2) => 3×3 reshape(view(::Vector{ComplexF64}, 6:14), 3, 3) with eltype
ComplexF64:
0.0+0.0im  1.41421+0.0im      0.0+0.0im
0.0+0.0im      0.0+0.0im  1.41421+0.0im
0.0+0.0im      0.0+0.0im      0.0+0.0im

*   …   [output of 8 more block(s) truncated]
```

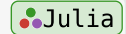


```julia
a⁻a⁺ = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
for (s, f) in fusiontrees(a⁻a⁺)
    if s.uncoupled[1] == only(f.uncoupled[1] ⊗ U1Irrep(-1)) && s.uncoupled[2] ==
    only(f.uncoupled[2] ⊗ U1Irrep(1))
        a⁻a⁺[s, f] .= sqrt(f.uncoupled[1].charge * s.uncoupled[2].charge)
    end
end
a⁻a⁺
```

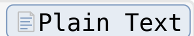


```
6×6←6×6 TensorMap{ComplexF64, Rep[U₁], 2, 2, Vector{ComplexF64}}:
codomain: (Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1) ⊗ Rep[U₁](0 =>
1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1))
domain: (Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1) ⊗ Rep[U₁](0 =>
1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1))
blocks:
* Irrep[U₁](0) => 1×1 reshape(view(::Vector{ComplexF64}, 1:1), 1, 1) with eltype
ComplexF64:
0.0 + 0.0im

* Irrep[U₁](1) => 2×2 reshape(view(::Vector{ComplexF64}, 2:5), 2, 2) with eltype
ComplexF64:
0.0+0.0im  0.0+0.0im
1.0+0.0im  0.0+0.0im
```

```
 * Irrep[U₁](2) => 3×3 reshape(view(::Vector{ComplexF64}, 6:14), 3, 3) with eltype
 ComplexF64:
     0.0+0.0im      0.0+0.0im  0.0+0.0im
 1.41421+0.0im      0.0+0.0im  0.0+0.0im
     0.0+0.0im  1.41421+0.0im  0.0+0.0im

 *   …   [output of 8 more block(s) truncated]
```



```julia
N = zeros(ComplexF64, V ← V)
for (s, f) in fusiontrees(N)
    N[s, f] .= f.uncoupled[1].charge
end
N
```



```
6←6 TensorMap{ComplexF64, Rep[U₁], 1, 1, Vector{ComplexF64}}:
 codomain: ⊗(Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1))
 domain: ⊗(Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1))
 blocks:
 * Irrep[U₁](0) => 1×1 reshape(view(::Vector{ComplexF64}, 1:1), 1, 1) with eltype
 ComplexF64:
 0.0 + 0.0im

 * Irrep[U₁](1) => 1×1 reshape(view(::Vector{ComplexF64}, 2:2), 1, 1) with eltype
 ComplexF64:
 1.0 + 0.0im

 * Irrep[U₁](2) => 1×1 reshape(view(::Vector{ComplexF64}, 3:3), 1, 1) with eltype
 ComplexF64:
 2.0 + 0.0im

 * Irrep[U₁](3) => 1×1 reshape(view(::Vector{ComplexF64}, 4:4), 1, 1) with eltype
 ComplexF64:
 3.0 + 0.0im

 * Irrep[U₁](4) => 1×1 reshape(view(::Vector{ComplexF64}, 5:5), 1, 1) with eltype
 ComplexF64:
 4.0 + 0.0im

 * Irrep[U₁](5) => 1×1 reshape(view(::Vector{ComplexF64}, 6:6), 1, 1) with eltype
 ComplexF64:
 5.0 + 0.0im
```

By inspecting the `subblocks` of each of these tensors you can directly verify that they each have the correct reduced tensor elements.

Creation and annihilation operators as symmetric tensors

Just as in the $\mathbb{Z}_2$ case, it is obvious that we cannot directly construct the creation and annihilation operators as instances of a `TensorMap(..., V ← V)` since they are not invariant under conjugation by the symmetry operator. However, it is possible to construct them as `TensorMap` s using an *auxiliary vector space*, based on the following intuition. The creation operator a^+ violates particle number conservation by mapping the occupation number n to $n+1$. From the point of view of representation theory, this process can be thought of as the *fusion* of an `U1Irrep(n)` with an `U1Irrep(1)` , naturally giving the fusion product `U1Irrep(n + 1)` . This means we can represent a^+ as a `TensorMap(..., V ← V ⊗ A)` , where the auxiliary vector space `A` contains the $+1$ irrep with degeneracy 1, `A = U1Space(1 => 1)` . Similarly, the decrease in occupation number when acting with a^- can be thought of as the *splitting* of an `U1Irrep(n)` into an `U1Irrep(n - 1)` and an `U1Irrep(1)` , leading to a representation in terms of a `TensorMap(. ., A ⊗ V ← V)` . Based on these observations, we can represent the matrix elements \eqref{eq:bosonopmatel} as subblocks labeled by the U_1 fusion trees

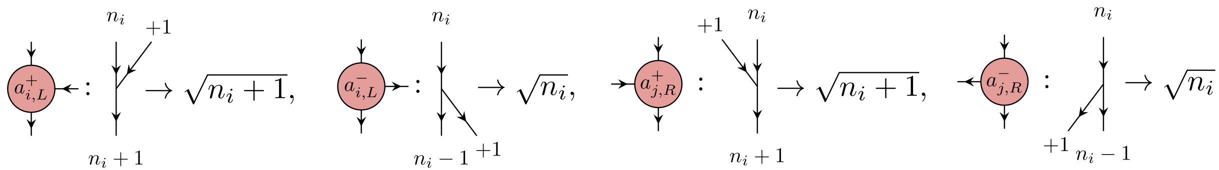


Figure 17.5: bosonops

We can then combine these operators to get the appropriate Hamiltonian terms,

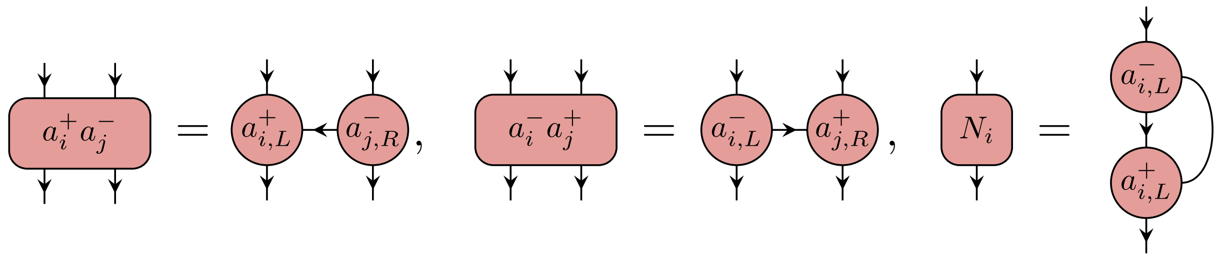


Figure 17.6: bosonham

Note

Although we have made a suggestive distinction between the 'left' and 'right' versions of the operators $a^\pm_L$ and $a^\pm_R$, one can actually be obtained from the other by permuting the physical and auxiliary indices of the corresponding `TensorMap` s. This permutation has no effect on the actual subblocks of the tensors due to the Abelian `FusionStyle` and bosonic `BraidingStyle` of U_1 irreps, so the left and right operators can in essence be seen as the 'same' tensors. This is no longer the case when considering non-Abelian symmetries, or symmetries associated with fermions or anyons. For these cases, permuting indices can in fact change the subblocks, as we will see next. As a consequence, it is much less clear how to construct two-site symmetric operators in terms of local symmetric objects.

The explicit construction then looks something like

```julia
A = U1Space(1 => 1)
```

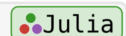
```
Rep[U₁](…) of dim 1:
 1 => 1
```

```julia
a⁺ = zeros(ComplexF64, V ← V ⊗ A)
```

```
for (s, f) in fusiontrees(a⁺)
    a⁺[s, f] .= sqrt(f.uncoupled[1].charge+1)
end
a⁺
```

Plain Text

```
6←6×1 TensorMap{ComplexF64, Rep[U₁], 1, 2, Vector{ComplexF64}}:
 codomain: ⊗(Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1))
 domain: (Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1) ⊗ Rep[U₁](1 =>
 1))
 blocks:
 * Irrep[U₁](1) => 1×1 reshape(view(::Vector{ComplexF64}, 1:1), 1, 1) with eltype
 ComplexF64:
 1.0 + 0.0im

 * Irrep[U₁](2) => 1×1 reshape(view(::Vector{ComplexF64}, 2:2), 1, 1) with eltype
 ComplexF64:
 1.4142135623730951 + 0.0im

 * Irrep[U₁](3) => 1×1 reshape(view(::Vector{ComplexF64}, 3:3), 1, 1) with eltype
 ComplexF64:
 1.7320508075688772 + 0.0im

 * Irrep[U₁](4) => 1×1 reshape(view(::Vector{ComplexF64}, 4:4), 1, 1) with eltype
 ComplexF64:
 2.0 + 0.0im

 * Irrep[U₁](5) => 1×1 reshape(view(::Vector{ComplexF64}, 5:5), 1, 1) with eltype
 ComplexF64:
 2.23606797749979 + 0.0im
```



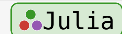
```
a⁻ = zeros(ComplexF64, A ⊗ V ← V)
for (s, f) in fusiontrees(a⁻)
    a⁻[s, f] .= sqrt(f.uncoupled[1].charge)
end
a⁻
```

Plain Text

```
1×6←6 TensorMap{ComplexF64, Rep[U₁], 2, 1, Vector{ComplexF64}}:
 codomain: (Rep[U₁](1 => 1) ⊗ Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 =>
 1))
 domain: ⊗(Rep[U₁](0 => 1, 1 => 1, 2 => 1, 3 => 1, 4 => 1, 5 => 1))
 blocks:
 * Irrep[U₁](1) => 1×1 reshape(view(::Vector{ComplexF64}, 1:1), 1, 1) with eltype
 ComplexF64:
 1.0 + 0.0im

 * Irrep[U₁](2) => 1×1 reshape(view(::Vector{ComplexF64}, 2:2), 1, 1) with eltype
 ComplexF64:
```

```
1.4142135623730951 + 0.0im

* Irrep[U₁](3) => 1×1 reshape(view(::Vector{ComplexF64}, 3:3), 1, 1) with eltype
ComplexF64:
1.7320508075688772 + 0.0im

* Irrep[U₁](4) => 1×1 reshape(view(::Vector{ComplexF64}, 4:4), 1, 1) with eltype
ComplexF64:
2.0 + 0.0im

* Irrep[U₁](5) => 1×1 reshape(view(::Vector{ComplexF64}, 5:5), 1, 1) with eltype
ComplexF64:
2.23606797749979 + 0.0im
```

It is then simple to check that this is indeed what we expect.




```
@tensor a⁺a⁻_bis[-1 -2; -3 -4] := a⁺[-1; -3 1] * a⁻[1 -2; -4]
@tensor a⁻a⁺_bis[-1 -2; -3 -4] := a⁻[1 -1; -3] * a⁺[-2; -4 1]
@tensor N_bis[-1 ; -2] := a⁺[-1; 1 2] * a⁻[2 1; -2]

@test a⁺a⁻_bis ≈ a⁺a⁻ atol=1e-14
@test a⁻a⁺_bis ≈ a⁻a⁺ atol=1e-14
@test N_bis ≈ N atol=1e-14
```

Plain Text

```
Test Passed
```

Note

From the construction of the Hamiltonian operators in terms of creation and annihilation operators we clearly see that they are invariant under a transformation $a^{\pm} \to e^{\pm i\theta} a^{\pm}$. More generally, for a two-site operator that is defined as the contraction of two one-site operators across an auxiliary space, modifying the one-site operators by applying transformations Q and Q^{-1} on their respective auxiliary spaces for any invertible Q leaves the resulting contraction unchanged. This ambiguity in the definition clearly shows that one should really always think in terms of the fully symmetric procucts of a^{+} and a^{-} rather than in terms of these operators themselves. In particular, one can always decompose such a symmetric product into the form above by means of an SVD.

17.4 Level 3: Fermions and the Kitaev model

While we have already covered quite a lot of ground towards understanding symmetric tensors in terms of fusion trees and corresponding subblocks, the symmetries considered so far have been quite 'simple' in the sense that sectors corresponding to irreps of $\mathbb{Z}_2$ and U_1 have *Abelian fusion rules* and *bosonic exchange statistics*. This means that the fusion of two irreps always gives a unique irrep as the fusion product, and that exchanging two irreps in a tensor product is trivial. In practice, this implies that for tensors with these symmetries the fusion trees are completely fixed by the uncoupled charges, which uniquely define both the inner lines and the coupled charge, and that tensor indices can be permuted freely without any strange side effects.

In the following we will consider examples with fermionic and even anyonic exchange statistics, and non-Abelian fusion rules. In going through these examples it will become clear that the fusion trees labeling the subblocks of a symmetric tensor imply more information than just a labeling.

Fermion parity symmetry

As a simple example we will consider the Kitaev chain, which describes a chain of interacting spinless fermions with nearest-neighbor hopping and pairing terms. The Hamiltonian of this model is given by

$$H = \sum_{\langle i,j \rangle} \left(-\frac{t}{2}(c_i^+ c_j^- - c_i^- c_j^+) + \frac{\Delta}{2}(c_i^+ c_j^+ - c_i^- c_j^-) \right) - \mu \sum_i N_i$$

where $N_i = c_i^+ c_i^-$ is the local particle number operator. As opposed to the previous case, the fermionic creation and annihilation operators now satisfy the anticommutation relations

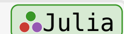
$$\begin{gathered} \left\{ c_i^-, c_j^- \right\} = \left\{ c_i^+, c_j^+ \right\} = 0 \\ \left\{ c_i^-, c_j^+ \right\} = \delta_{ij}. \end{gathered}$$

These relations justify the choice of the relative minus sign in the hopping and pairing terms. Indeed, since fermionic operators on different sites always anticommute, these relative minus signs are needed to ensure that the Hamiltonian is Hermitian, since $\left(c_i^+ c_j^-\right)^\dagger = c_j^+ c_i^- = -c_i^- c_j^+$ and $\left(c_i^+ c_j^+\right)^\dagger = c_j^- c_i^- = -c_i^- c_j^-$. The anticommutation relations also naturally restrict the local occupation number to be 0 or 1, leading to a well-defined notion of *fermion-parity*. The local fermion-parity operator is related to the fermion number operator as $Q_i = (-1)^{N_i}$, and is diagonal in the occupation number basis. The Hamiltonian \eqref{eq:kitaev} is invariant under conjugation by the global fermion-parity operator, $QHQ^\dagger = H$, where

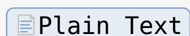
$$Q = \exp\left(i\pi \sum_i N_i \right) = (-1)^{\sum_i N_i}.$$

This fermion parity symmetry, which we will denote as $f\mathbb{Z}_2$, is a $\mathbb{Z}_2$-like symmetry in the sense that it has a trivial representation, which we call *even* and again denote by '0', and a sign representation which we call *odd* and denote by '1'. The fusion rules of these irreps are the same as for $\mathbb{Z}_2$. Similar to the previous case, the local symmetry operator Q_i is already diagonal, so the occupation number basis coincides with the irrep basis and we don't need an additional basis transform. The important difference with a regular $\mathbb{Z}_2$ symmetry is that the irreps of $f\mathbb{Z}_2$ have fermionic braiding statistics, in the sense that exchanging two odd irreps gives rise to a minus sign.

In TensorKit.jl, an $f\mathbb{Z}_2$-graded vector spaces is represented as a `Vect[FermionParity]` space, where a given $f\mathbb{Z}_2$ irrep can be represented as a `FermionParity` sector instance. Using the simplest instance of a vector space containing a single even and odd irrep, we can already demonstrate the corresponding fermionic braiding behavior by performing a permutation on a simple `TensorMap`.

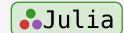


```
V = Vect[FermionParity](0 => 1, 1 => 1)
t = ones(ComplexF64, V ← V ⊗ V)
subblocks(t)
```

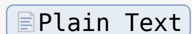


```
subblocks(::TensorMap{ComplexF64, Vect[FermionParity], 1, 2,
Vector{ComplexF64}}):
```

```
 * (FermionParity(0),) ← (FermionParity(0), FermionParity(0)) => 1×1×1
 StridedViews.StridedView{ComplexF64, 3, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1] =
 1.0 + 0.0im

 * (FermionParity(0),) ← (FermionParity(1), FermionParity(1)) => 1×1×1
 StridedViews.StridedView{ComplexF64, 3, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1] =
 1.0 + 0.0im

 * (FermionParity(1),) ← (FermionParity(1), FermionParity(0)) => 1×1×1
 StridedViews.StridedView{ComplexF64, 3, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1] =
 1.0 + 0.0im

 * (FermionParity(1),) ← (FermionParity(0), FermionParity(1)) => 1×1×1
 StridedViews.StridedView{ComplexF64, 3, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1] =
 1.0 + 0.0im
```



```julia
tp = permute(t, ((1,), (3, 2)))
subblocks(tp)
```



```
subblocks(::TensorMap{ComplexF64, Vect[FermionParity], 1, 2,
Vector{ComplexF64}}):
 * (FermionParity(0),) ← (FermionParity(0), FermionParity(0)) => 1×1×1
 StridedViews.StridedView{ComplexF64, 3, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1] =
 1.0 + 0.0im

 * (FermionParity(0),) ← (FermionParity(1), FermionParity(1)) => 1×1×1
 StridedViews.StridedView{ComplexF64, 3, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1] =
 -1.0 + 0.0im

 * (FermionParity(1),) ← (FermionParity(1), FermionParity(0)) => 1×1×1
 StridedViews.StridedView{ComplexF64, 3, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1] =
 1.0 + 0.0im

 * (FermionParity(1),) ← (FermionParity(0), FermionParity(1)) => 1×1×1
 StridedViews.StridedView{ComplexF64, 3, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1] =
 1.0 + 0.0im
```

In other words, when exchanging the two domain vector spaces, the reduced tensor elements of the `TensorMap` for which both uncoupled irreps in the domain of the corresponding fusion tree are odd picks up a minus sign, exactly as we would expect for fermionic charges.

Constructing the Hamiltonian

We can directly construct the Hamiltonian terms as symmetric `TensorMap` s using the same procedure as before starting from their matrix elements in the occupation number basis. However, in this case we should be a bit more careful about the precise definition of the basis states in composite systems. Indeed, the tensor product structure of fermionic systems is inherently tricky to deal with, and should ideally be treated in the context of *super vector spaces*. For two sites, we can define the following basis states on top of the fermionic vacuum $|00\rangle$:

$$\begin{aligned} |01\rangle &= c_2^+ \ |00\rangle, \\ |10\rangle &= c_1^+ \ |00\rangle, \\ |11\rangle &= c_1^+ c_2^+ \ |00\rangle. \end{aligned}$$

This definition in combination with the anticommutation relations above give rise to the nonzero matrix elements

$$\begin{aligned} c_1^+ c_2^- \ |0,1\rangle &= |1,0\rangle, \\ c_1^- c_2^+ \ |1,0\rangle &= - \ |0,1\rangle, \\ c_1^+ c_2^+ \ |0,0\rangle &= |1,1\rangle, \\ c_1^- c_2^- \ |1,1\rangle &= - \ |0,0\rangle, \\ N \ |n\rangle &= n \ |n\rangle. \end{aligned}$$

While the signs in these expressions may seem a little unintuitive at first sight, they are essential to the fermionic nature of the system. Indeed, if we for example work out the matrix element of $c_1^- c_2^+$ we find

$$c_1^- c_2^+ \ |1,0\rangle = c_1^- c_2^+ c_1^+ \ |0,0\rangle = -c_2^+ c_1^- c_1^+ \ |0,0\rangle = -c_2^+ (\mathbb{1} - c_1^+ c_1^-) \ |0,0\rangle = -c_2^+ \ |0,0\rangle = - \ |0,1\rangle.$$

Once we have these matrix elements the hard part is done, and we can straightforwardly associate these to the following $f\mathbb{Z}_2$ fusion trees with corresponding reduced tensor elements,

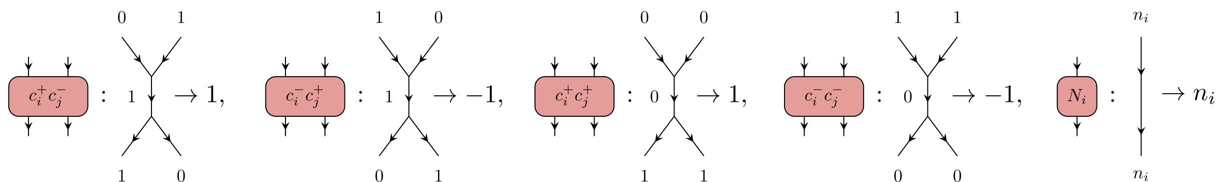


Figure 17.7: fZ2_fusiontrees

Given this information, we can go through the same procedure again to construct c^+c^-, c^-c^+ and N operators as `TensorMap` s over $f\mathbb{Z}_2$-graded vector spaces.

```julia
V = Vect[FermionParity](0 => 1, 1 => 1)
```

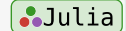
```
Vect[FermionParity](…) of dim 2:
 0 => 1
 1 => 1
```

```julia
c⁺c⁻ = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
odd = FermionParity(1)
for (s, f) in fusiontrees(c⁺c⁻)
```

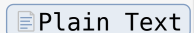
```
    if s.uncoupled[1] == odd && f.uncoupled[2] == odd && f.coupled == odd
        c⁺c⁻[s, f] .= 1
    end
end
subblocks(c⁺c⁻)
```

```
subblocks(::TensorMap{ComplexF64, Vect[FermionParity], 2, 2,
Vector{ComplexF64}}):
 * (FermionParity(0), FermionParity(0)) ← (FermionParity(0), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(1)) ← (FermionParity(0), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(0), FermionParity(0)) ← (FermionParity(1), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(1)) ← (FermionParity(1), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(0)) ← (FermionParity(1), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(0), FermionParity(1)) ← (FermionParity(1), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(0)) ← (FermionParity(0), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
```

```
 1.0 + 0.0im

 * (FermionParity(0), FermionParity(1)) ← (FermionParity(0), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im
```

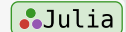


```julia
c⁻c⁺ = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
for (s, f) in fusiontrees(c⁻c⁺)
    if f.uncoupled[1] == odd && s.uncoupled[2] == odd && f.coupled == odd
        c⁻c⁺[s, f] .= -1
    end
end
subblocks(c⁻c⁺)
```

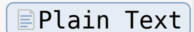


```
subblocks(::TensorMap{ComplexF64, Vect[FermionParity], 2, 2,
Vector{ComplexF64}}):
 * (FermionParity(0), FermionParity(0)) ← (FermionParity(0), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(1)) ← (FermionParity(0), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(0), FermionParity(0)) ← (FermionParity(1), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(1)) ← (FermionParity(1), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(0)) ← (FermionParity(1), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im
```

```
 * (FermionParity(0), FermionParity(1)) ← (FermionParity(1), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 -1.0 + 0.0im

 * (FermionParity(1), FermionParity(0)) ← (FermionParity(0), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(0), FermionParity(1)) ← (FermionParity(0), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im
```

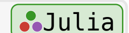


```julia
c⁺c⁺ = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
odd = FermionParity(1)
for (s, f) in fusiontrees(c⁺c⁺)
    if s.uncoupled[1] == odd && f.uncoupled[1] != odd && f.coupled != odd
        c⁺c⁺[s, f] .= 1
    end
end
subblocks(c⁺c⁺)
```

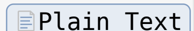


```
subblocks(::TensorMap{ComplexF64, Vect[FermionParity], 2, 2,
Vector{ComplexF64}}):
 * (FermionParity(0), FermionParity(0)) ← (FermionParity(0), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(1)) ← (FermionParity(0), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 1.0 + 0.0im

 * (FermionParity(0), FermionParity(0)) ← (FermionParity(1), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im
```

```
 * (FermionParity(1), FermionParity(1)) ← (FermionParity(1), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(0)) ← (FermionParity(1), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(0), FermionParity(1)) ← (FermionParity(1), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(0)) ← (FermionParity(0), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(0), FermionParity(1)) ← (FermionParity(0), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im
```

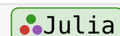


```
c⁻c⁻ = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
for (s, f) in fusiontrees(c⁻c⁻)
    if s.uncoupled[1] != odd && f.uncoupled[2] == odd && f.coupled != odd
        c⁻c⁻[s, f] .= -1
    end
end
subblocks(c⁻c⁻)
```



```
subblocks(::TensorMap{ComplexF64, Vect[FermionParity], 2, 2,
Vector{ComplexF64}}):
 * (FermionParity(0), FermionParity(0)) ← (FermionParity(0), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im
```

```
 * (FermionParity(1), FermionParity(1)) ← (FermionParity(0), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(0), FermionParity(0)) ← (FermionParity(1), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 -1.0 + 0.0im

 * (FermionParity(1), FermionParity(1)) ← (FermionParity(1), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(0)) ← (FermionParity(1), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(0), FermionParity(1)) ← (FermionParity(1), FermionParity(0)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(1), FermionParity(0)) ← (FermionParity(0), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FermionParity(0), FermionParity(1)) ← (FermionParity(0), FermionParity(1)) =>
 1×1×1×1 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64},
 typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im
```



```julia
N = zeros(ComplexF64, V ← V)
for (s, f) in fusiontrees(N)
    N[s, f] .= f.coupled == odd ? 1 : 0
end
subblocks(N)
```

```
subblocks(::TensorMap{ComplexF64, Vect[FermionParity], 1, 1,
Vector{ComplexF64}}):
 * (FermionParity(0),) ← (FermionParity(0),) => 1×1
StridedViews.StridedView{ComplexF64, 2, Memory{ComplexF64}, typeof(identity)}:
 0.0 + 0.0im

 * (FermionParity(1),) ← (FermionParity(1),) => 1×1
StridedViews.StridedView{ComplexF64, 2, Memory{ComplexF64}, typeof(identity)}:
 1.0 + 0.0im
```

We can easily all the reduced tensor elements are indeed correct.

Note

Working with fermionic systems is inherently tricky, as can already be seen from something as simple as computing matrix elements of fermionic operators. Similarly, while constructing symmetric tensors that correspond to the symmetric Hamiltonian terms was still quite straightforward, it is far less clear in this case how to construct these terms as contractions of local symmetric tensors representing individual creation and annihilation operators. While such a decomposition can always be in principle obtained using a (now explicitly fermionic) SVD, manually constructing such tensors as we did in the bosonic case is far from trivial. Trying this would be a good exercise in working with fermionic symmetries, but it is not something we will do here.

17.5 Level 4: Non-Abelian symmetries and the quantum Heisenberg model

We will now move on to systems which have more complicated *non-Abelian* symmetries. For a non-Abelian symmetry group G, the fact that its elements do not all commute has a profound impact on its representation theory. In particular, the irreps of such a group can be higher dimensional, and the fusion of two irreps can give rise to multiple different irreps. On the one hand, this means that fusion trees of these irreps are no longer completely determined by the uncoupled charges. Indeed, in this case some of the internal structure of the `FusionTree` type we have ignored before will become relevant (of which we will give an example below). On the other hand, it follows that fusion trees of irreps now not only label reduced tensor elements, but also encode a certain *nontrivial symmetry structure*. We will make this statement more precise in the following, but the fact that this is necessary is quite intuitive. If we recall our original statement that symmetric tensors consist of subblocks associated to fusion trees which carry irrep labels, then for higher-dimensional irreps the corresponding fusion trees must encode some additional information that implicitly takes into account the internal structure of the representation spaces. In particular, this means that the conversion of an operator, given its matrix elements in the irrep basis, to the subblocks of the corresponding symmetric `TensorMap` is less straightforward since it requires an understanding of exactly what this implied internal structure is. Therefore, we require some more discussion before we can actually move on to an example.

We'll start by discussing the general structure of a `TensorMap` which is symmetric under a non-Abelian group symmetry. We then given an example based on SU_2, where we construct the Heisenberg Hamiltonian using two different approaches. Finally, we show how the more intuitive approach can be used to obtain an elegant generalization to the SU_N-symmetric case.

Block sparsity revisited: the Wigner-Eckart theorem

Let us recall some basics of representation theory first. Consider a group G and a corresponding representation space V, such that every element $g \in G$ can be realized as a unitary operator $U_g : V \to V$. Let h be a `TensorMap` whose domain and codomain are given by the tensor product of two of these representation spaces. By definition, the statement that 'h is symmetric under G' means that

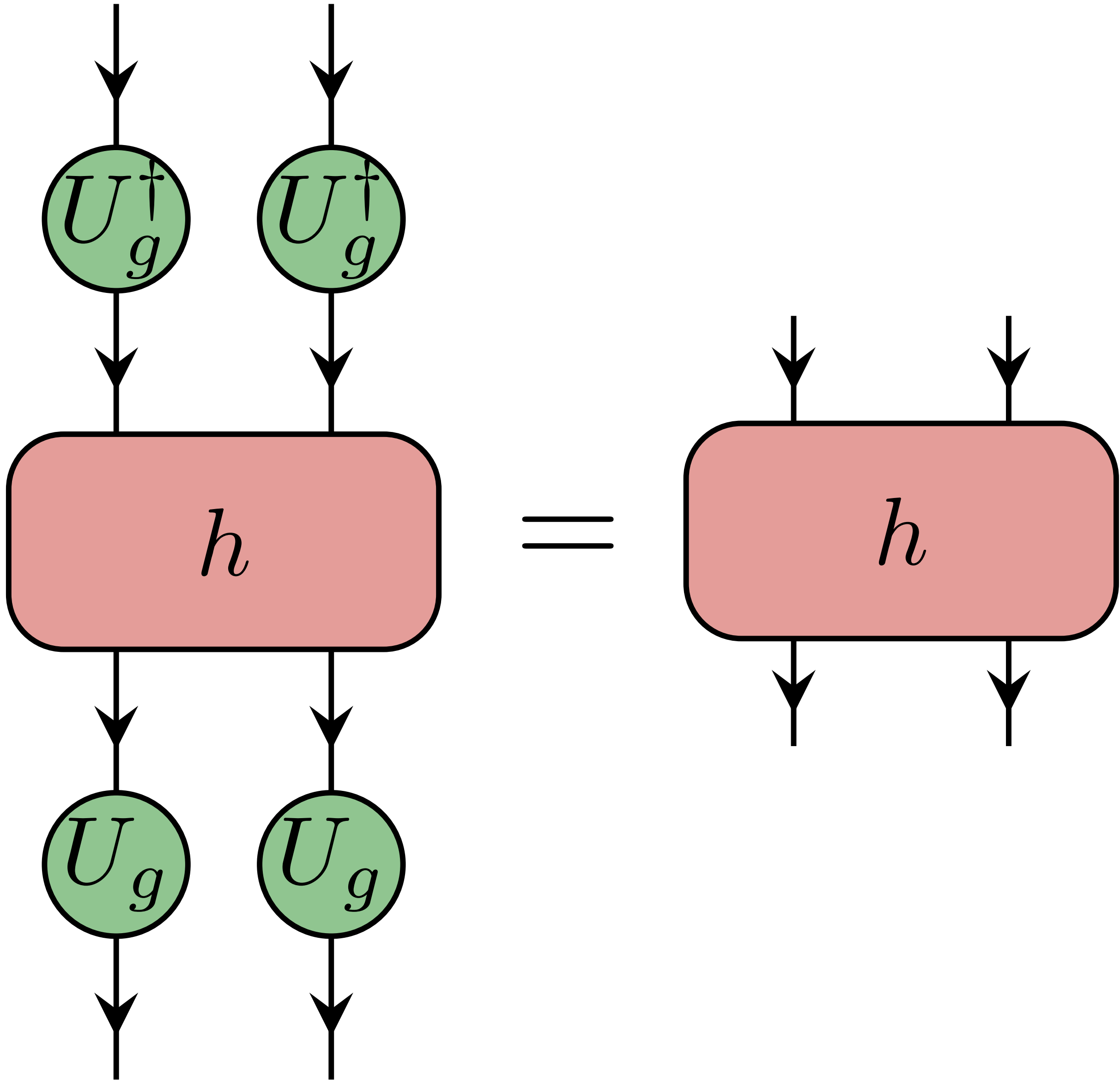


Figure 17.8: symmetric_tensor

for every $g \in G$. If we label the irreducible representations of G by l, then any representation space can be decomposed into a direct sum of irreducible representations, $V = \bigoplus_l V^{(l)}$, in such a way that U_g is block-diagonal where each matrix block is labeled by a particular irrep l. For each irrep space $V^{(l)}$ we can define an orthonormal basis labeled as $|l, m\rangle$, where the auxiliary label m can take $\dim\left(V^{(l)}\right)$ different values. Since we know that tensors are multilinear maps over tensor product spaces, it is natural to consider the tensor product of representation spaces in more detail.

From the representation theory of groups, it is known that the product of two irreps can in turn be decomposed into a direct sum of irreps, $V^{(l_1)} \otimes V^{(l_2)} \cong \bigoplus_k V^{(k)}$. The precise nature of this decomposition, also refered to as the *Clebsch-Gordan problem*, is given by the so-called *Clebsch-Gordan coefficients*, which we will denote as $C^k_{l_1,l_2}$. This set of coefficients, which can be interpreted as a $\dim\left(V^{(l_1)}\right) \times \dim\left(V^{(l_2)}\right) \times \dim\left(V^{(k)}\right)$ array, encodes how a basis state $|k, n\rangle \in V^{(k)}$ corresponding

to some term in the direct sum can be decomposed into a linear combination of basis vectors $|l_1, m_1\rangle \otimes |l_2, m_2\rangle$ of the tensor product space:

$$|k, n\rangle = \sum_{m_1, m_2} \left(C^{k}_{l_1, l_2}\right)^{n}_{m_1, m_2} |l_1, m_1\rangle \otimes |l_2, m_2\rangle.$$

These recoupling coefficients turn out to be essential to the structure of symmetric tensors, which can be best understood in the context of the Wigner-Eckart theorem. This theorem implies that for any `TensorMap` h that is symmetric under G, its matrix elements in the tensor product irrep basis are given by the product of Clebsch-Gordan coefficients which characterize the coupling of the basis states in the domain and codomain, and a so-called *reduced tensor element* which only depends on the irrep labels. Concretely, the matrix element $\langle l_1, m_1| \otimes \langle l_2, m_2|\; h\; |l_3, m_3\rangle \otimes |l_4, m_4\rangle$ is given by

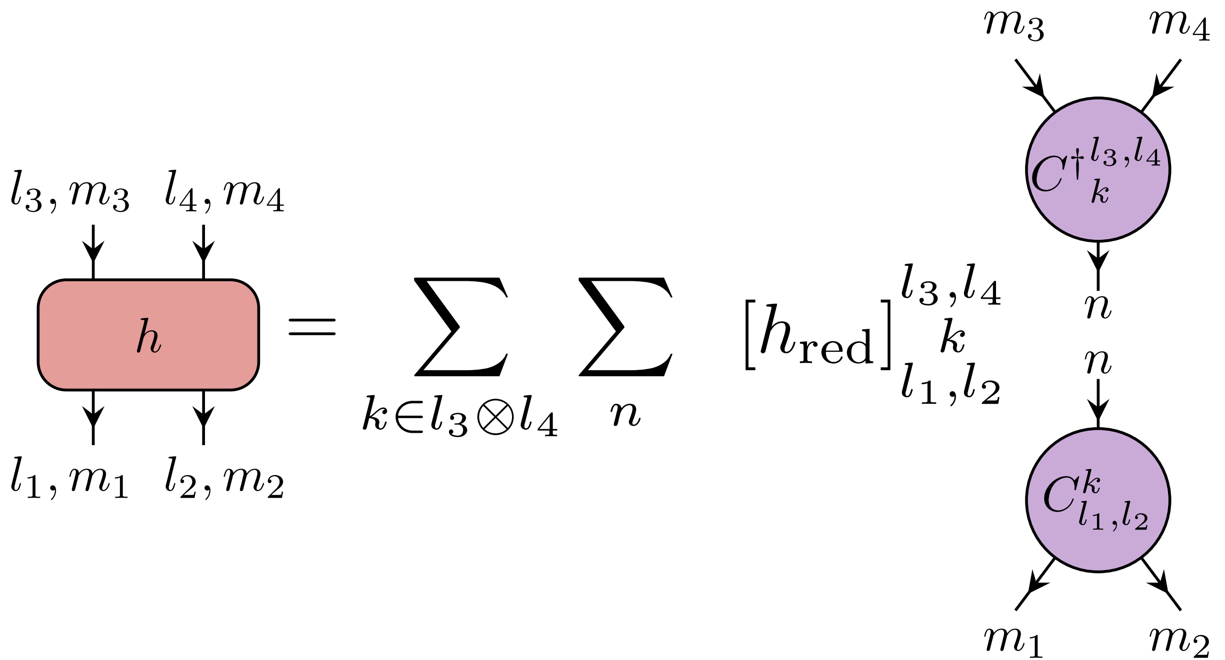


Figure 17.9: wignereckart

Here, the sum runs over all possible irreps k in the fusion product $l_3 \otimes l_4$ and over all basis states $|k, n\rangle$ of $V^{(k)}$. The reduced tensor elements h_{red} are independent of the basis state labels and only depend on the irrep labels themselves. Each reduced tensor element should be interpreted as being labeled by an irrep fusion tree,

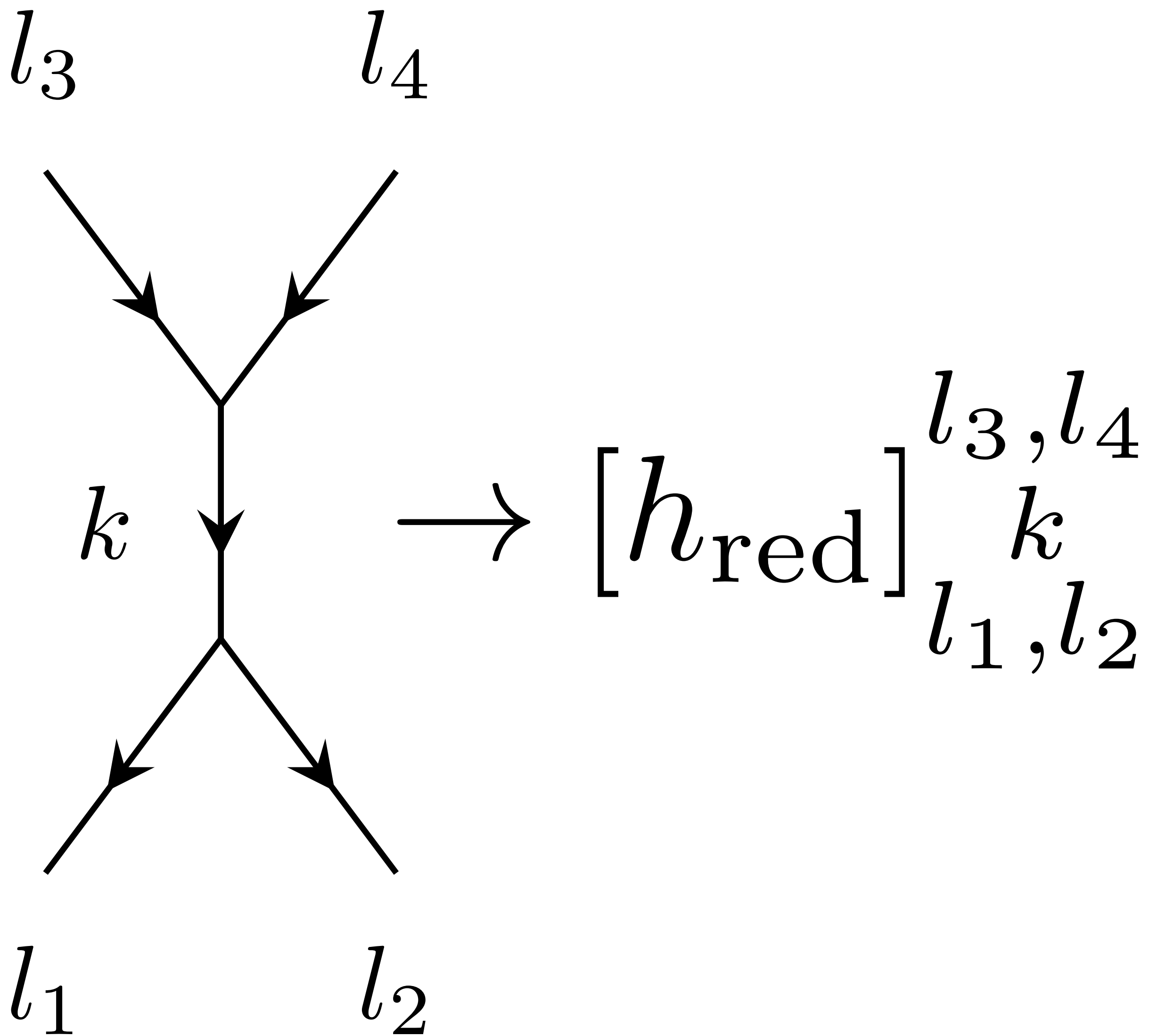


Figure 17.10: anotherfusiontree

The fusion tree itself in turn implies the Clebsch-Gordan coefficients $C^{k}_{l_1,l_2}$ and conjugate coefficients $C^{\dagger}_{(k)^{l_1,l_2}}$ encode the splitting (decomposition) of the coupled basis state $|k,n\rangle$ to the codomain basis states $|l_1,m_1\rangle \otimes |l_2,m_2\rangle$ and the coupling of the domain basis states $|l_3,m_3\rangle \otimes |l_4,m_4\rangle$ to the coupled basis state $|k,n\rangle$ respectively.

The Wigner-Eckart theorem dictates that this structure in terms of Clebsch-Gordan coefficients is necessary to ensure that the corresponding tensor is symmetric. It is precisely this structure that is inherently encoded into the fusion tree part of a symmetric `TensorMap` . In particular, **the subblock value associated to each fusion tree in a symmetric tensor is precisely the reduced tensor element in the Clebsch-Gordan decomposition**.

Note

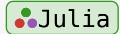
In the Clebsch-Gordan decomposition given above, our notation has actually silently assumed that each irrep k only occurs once in the fusion product of the uncoupled irreps l_1 and l_2. However, there exist symmetries which have **fusion multiplicities**, where two irreps can fuse to a given coupled irrep in multiple *distinct* ways. In TensorKit.jl, these correspond to `Sector` types with a `GenericFusion <: FusionStyle` fusion style. In the presence of fusion multiplicities, the Clebsch-

Gordan coefficients actually have an additional index which labels the particular fusion channel according to which l_1 and l_2 fuse to k. Since the fusion of SU_2 irreps is multiplicity-free, we could safely ignore this nuance here. We will encounter the implication of fusion multiplicities shortly, and will consider an example of a symmetry which has these multiplicities below.

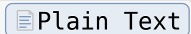
As a small demonstration of this fact, we can make a simple SU_2-symmetric tensor with trivial subblock values and verify that its implied symmetry structure exactly corresponds to the expected Clebsch-Gordan coefficient. First, we recall that the irreps of SU_2 can be labeled by a halfinteger *spin* that takes values $l = 0, \frac{1}{2}, 1, \frac{3}{2}, \ldots$, and where the dimension of the spin-l representation is equal to $2l + 1$. The fusion rules of SU_2 are given by

$$l_1 \otimes l_2 \cong \bigoplus_{k=|l_1-l_2|}^{l_1+l_2} k.$$

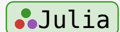
These are clearly non-Abelian since multiple terms appear on the right hand side, for example $\frac{1}{2} \otimes \frac{1}{2} \cong 0 \oplus 1$. In TensorKit.jl, a SU_2-graded vector space is represented as an `SU2Space` , where a given SU_2 irrep can be represented as an `SU2Irrep` instance of integer or halfinteger spin as encoded in its `j` field. If we construct a `TensorMap` whose symmetry structure corresponds to the coupling of two spin-$\frac{1}{2}$ irreps to a spin-1 irrep in the sense of \eqref{eq:cg*decomposition}, we can then convert it to a plain array and compare it to the ``\mathsf{SU}2``* Clebsch-Gordan coefficients implemented in the WignerSymbols.jl package.



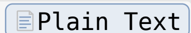
```julia
V1 = SU2Space(1//2 => 1)
V2 = SU2Space(1 => 1)
t = ones(ComplexF64, V1 ⊗ V1 ← V2)
```



```
2×2←3 TensorMap{ComplexF64, Rep[SU₂], 2, 1, Vector{ComplexF64}}:
 codomain: (Rep[SU₂](1/2 => 1) ⊗ Rep[SU₂](1/2 => 1))
 domain: ⊗(Rep[SU₂](1 => 1))
 blocks:
 * Irrep[SU₂](1) => 1×1 reshape(view(::Vector{ComplexF64}, 1:1), 1, 1) with eltype
 ComplexF64:
 1.0 + 0.0im
```

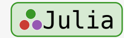


```julia
ta = convert(Array, t)
```



```
2×2×3 Array{ComplexF64, 3}:
[:, :, 1] =
 1.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im

[:, :, 2] =
      0.0+0.0im  0.707107+0.0im
 0.707107+0.0im       0.0+0.0im

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

The conversion gives us a $2 \times 2 \times 3$ array, which exactly corresponds to the size of the $C^{1}_{\frac{1}{2},\frac{1}{2}}$ Clebsch-Gordan array. In order to explicitly compare whether the entries match we need to know the ordering of basis states assumed by TensorKit.jl when converting the tensor to its matrix elements in the irrep basis. For SU_2 the irrep basis is ordered in ascending magnetic quantum number m, which gives us a map $m = i - (l + 1)$ for mapping an array index to a corresponding magnetic quantum number for the spin-l irrep.



```
checks = map(Iterators.product(1:dim(V1), 1:dim(V1), 1:dim(V2))) do (i1,
i2, i3)
    # map basis state index to magnetic quantum number
    m1 = i1 - (1//2 + 1)
    m2 = i2 - (1//2 + 1)
    m3 = i3 - (1 + 1)
    # check the corresponding array entry
    return ta[i1, i2, i3] ≈ clebschgordan(1//2, m1, 1//2, m2, 1, m3)
end
@test all(checks)
```

```
Test Passed
```

Plain Text

Based on this discussion, we can quantify the aforementioned 'difficulties' in the inverse operation of what we just demonstrated, namely converting a given operator to a symmetric `TensorMap` given only its matrix elements in the irrep basis. Indeed, it is now clear that this precisely requires isolating the reduced tensor elements introduced above. Given the matrix elements of the operator in the irrep basis, this can in general be done by solving the system of equations implied by the Clebsch-Gordan decomposition. A simpler way to achieve the same thing is to make use of the fact that the Clebsch-Gordan tensors form a complete orthonormal basis on the coupled space. Indeed, by projecting out the appropriate Clebsch-Gordan coefficients and using their orthogonality relations, we can construct a diagonal operator on each coupled irrep space $V^{(k)}$. Each of these diagonal operators is proportional to the identity, where the proportionality factor is precisely the reduced tensor element associated to the corresponding irrep fusion tree.

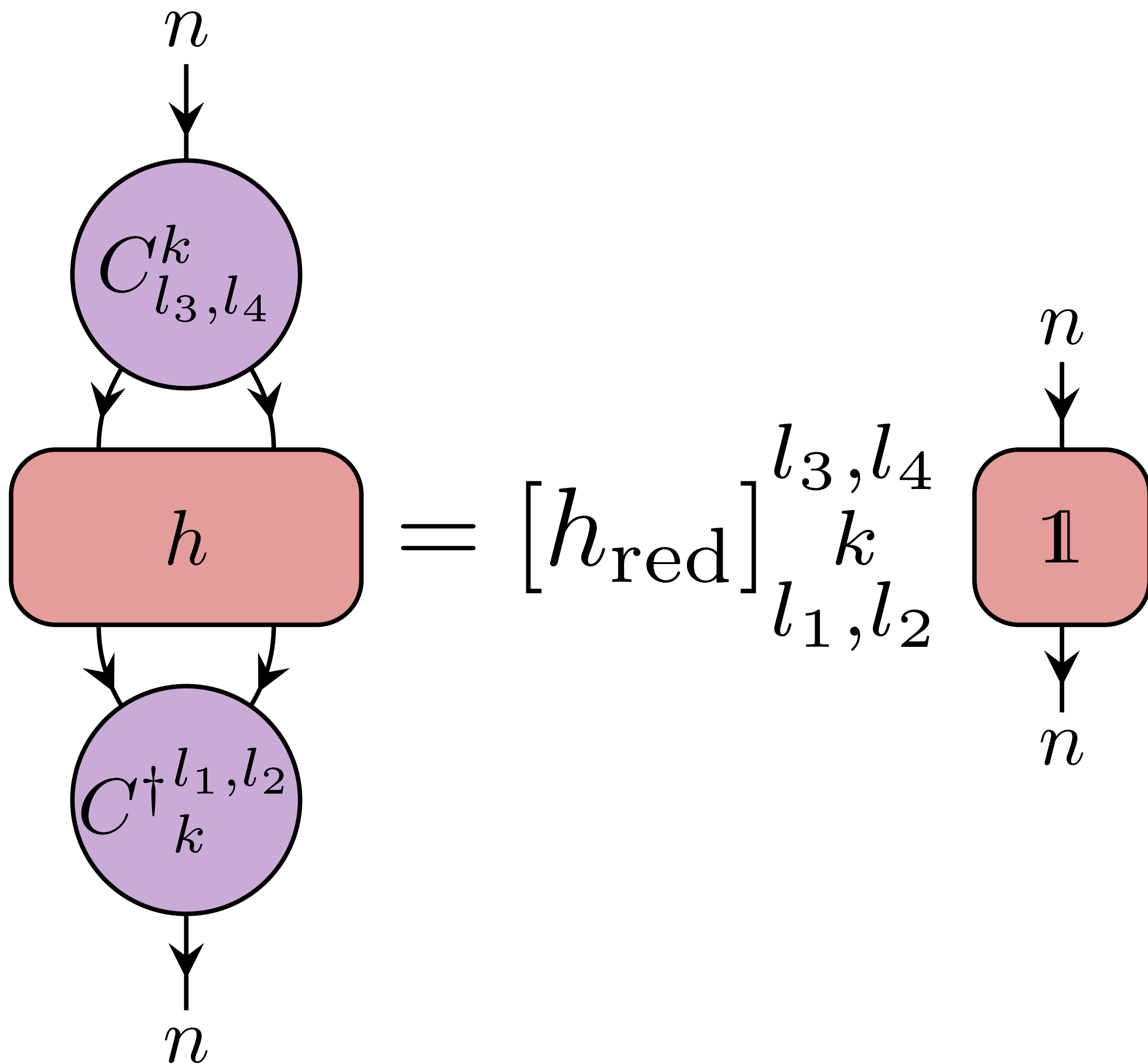
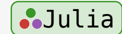


Figure 17.11: none2symm

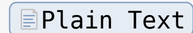
This procedure works for any group symmetry, and all we need are matrix elements of the operator in the irrep basis and the Clebsch-Gordan coefficients. In the following, we demonstrate this explicit procedure for the particular example of $G = SU_2$. However, it should be noted that, for other non-Abelian groups, the Clebsch-Gordan coefficients may not be as easy to compute (generically, no closed formulas exist). In addition, the procedure for manually projecting out the reduced tensor elements requires being particularly careful about the correspondence between the basis states used to define the original matrix elements and those implied by the Clebsch-Gordan coefficients. Finally, for some symmetries supported in TensorKit. l, there are simply no Clebsch-Gordan coefficients. Therefore, it is often easier and sometimes simply necessary to directly construct the symmetric tensor and then fill in its reduced tensor elements based on some representation theory. We will cover some examples of this below.

Having introduced and demonstrated the Clebsch-Gordan decomposition, the corresponding coefficients and their role in symmetric tensors for the example of SU_2 using the WignerSymbols.jl package, we now continue our discussion using only TensorKit.jl internals. Within TensorKit.jl, the $\dim\left(V^{(l_1)}\right) \times \dim\left(V^{(l_2)}\right) \times \dim\left(V^{(k)}\right)$ array of coefficients that encodes the splitting of the irrep space $V^{(k)}$ to the tensor product of irrep spaces $V^{(l_1)} \otimes V^{(l_2)}$ according to the Clebsch-Gordan decomposition \eqref{eq:cg_decomposition} above can be explicitly constructed by calling the

`TensorKitSectors.fusiontensor` method on the corresponding `Sector` instances, `fusiontensor(l₁, l₂, k)`. This `fusiontensor` is defined for any sector type corresponding to a symmetry which admits Clebsch-Gordan coefficients. For our example above, we can build the corresponding fusion tensor as



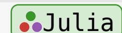
```julia
using TensorKit: fusiontensor
f = fusiontensor(SU2Irrep(1//2), SU2Irrep(1//2), SU2Irrep(1))
```



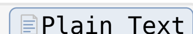
```
2×2×3×1 Array{Float64, 4}:
[:, :, 1, 1] =
 1.0  0.0
 0.0  0.0

[:, :, 2, 1] =
 0.0       0.707107
 0.707107  0.0

[:, :, 3, 1] =
 0.0  0.0
 0.0  1.0
```

We see that this fusion tensor has a size `2×2×3×1`, which contains an additional trailing `1` to what we might expect. In the general case, `fusiontensor` returns a four-dimensional array, where the size of the first three dimensions corresponds to the dimensions of the irrep spaces under consideration, and the last index lables the different fusion channels, where its dimension corresponds to the number of distinct ways the irreps l_1 and l_2 can fuse to irrep k. This is precicely the extra label of the Clebsch-Gordan coefficients that is required in the the presence of fusion multiplicities. Since SU_2 is multiplicity-free, we can just discard this last index here.

We can now explicitly verify that this `fusiontensor` indeed does what we expect it to do:



```julia
@test ta ≈ f[:, :, :, 1]
```



```
Test Passed
```

Of course, in this case `fusiontensor` just calls `Wignersymbols.clebschgordan` under the hood. However, `TensorKitSectors.fusiontensor` works for general symmetries, and makes it so that we never have to manually assemble the coefficients into an array.

The 'generic' approach to the spin-1 Heisenberg model: Wigner-Eckart in action

Consider the spin-1 Heisenberg model with Hamiltonian

$$H = J \sum_{\langle i,j \rangle} \vec{S}_i \cdot \vec{S}_j$$

where $\vec{S} = (S^x, S^y, S^z)$ are the spin operators. The physical Hilbert space at each site is the three-dimensional spin-1 irrep of SU_2. Each two-site exchange operator $\vec{S}_i \cdot \vec{S}_j$ in the sum commutes with a global transformation $g \in SU_2$, so that it satisfies the above symmetry condition. Therefore, we can represent it as an SU_2-symmetric `TensorMap`, as long as we can isolate its reduced tensor elements.

In order to apply the above procedure, we first require the matrix elements in the irrep basis. These can be constructed as a $3 \times 3 \times 3 \times 3$ array `SS` using the familiar representation of the SU_2 generators in the spin-1 representation, with respect to the $\{|1,-1\rangle, |1,0\rangle, |1,1\rangle\}$ basis.

```julia
Sx = 1 / sqrt(2) * ComplexF64[0 1 0; 1 0 1; 0 1 0]
Sy = 1 / sqrt(2) * ComplexF64[0 1im 0; -1im 0 1im; 0 -1im 0]
Sz = ComplexF64[-1 0 0; 0 0 0; 0 0 1]

@tensor SS_arr[-1 -2; -3 -4] := Sx[-1; -3] * Sx[-2; -4] + Sy[-1; -3] * Sy[-2; -4]
+ Sz[-1; -3] * Sz[-2; -4]
```

The next step is to project out the reduced tensor elements by taking the overlap with the appropriate Clebsch-Gordan coefficients. In our current case of a spin-1 physical space, we have $l_1 = l_2 = l_3 = l_4 = 1$, and the coupled irrep k can therefore take the values $0, 1, 2$. The reduced tensor element for a given k can be implemented in the following way:

```julia
function get_reduced_element(k::SU2Irrep)
    # construct Clebsch-Gordan coefficients for coupling 1 ⊗ 1 to k
    f = fusiontensor(SU2Irrep(1), SU2Irrep(1), k)[:, :, :, 1]
    # project out diagonal matrix on coupled irrep space
    @tensor reduced_matrix[-1; -2] := conj(f[1 2; -1]) * SS_arr[1 2; 3 4] * f[3 4;
    -2]
    # check that it is proportional to the identity
    @assert isapprox(reduced_matrix, reduced_matrix[1, 1] * I; atol=1e-12)
    # return the proportionality factor
    return reduced_matrix[1, 1]
end
```

```
get_reduced_element (generic function with 1 method)
```

If we use this to compute the reduced tensor elements for $k = 0, 1, 2$,

```julia
get_reduced_element(SU2Irrep(0))
```

```
-1.9999999999999993 + 0.0im
```

```julia
get_reduced_element(SU2Irrep(1))
```

```
-0.9999999999999997 + 0.0im
```

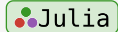
```julia
get_reduced_element(SU2Irrep(2))
```

```
1.0 + 0.0im
```

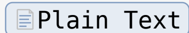
we can read off the entries

$$.01ex \left[(\vec{S}_i \cdot \vec{S}_j)_{\text{red}}\right]^{1,1}_{1,1} 0 = -2, \quad \left[(\vec{S}_i \cdot \vec{S}_j)_{\text{red}}\right]^{1,1}_{1,1} 1 = -1, \quad \left[(\vec{S}_i \cdot \vec{S}_j)_{\text{red}}\right]^{1,1}_{1,1} 2 = 1,$$

These can then be used to construct the symmetric `TensorMap` representing the exchange interaction:

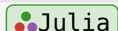


```julia
V = SU2Space(1 => 1)
SS = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
for (s, f) in fusiontrees(SS)
    k = f.coupled
    SS[s, f] .= get_reduced_element(k)
end
subblocks(SS)
```

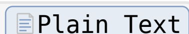


```
subblocks(::TensorMap{ComplexF64, Rep[SU₂], 2, 2,
Vector{ComplexF64}}):
 * (FusionTree{Irrep[SU₂]}((1, 1), 0, (false, false), ()), FusionTree{Irrep[SU₂]}
 ((1, 1), 0, (false, false), ())) => 1×1×1×1 StridedViews.StridedView{ComplexF64,
 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 -1.9999999999999993 + 0.0im

 * (FusionTree{Irrep[SU₂]}((1, 1), 1, (false, false), ()), FusionTree{Irrep[SU₂]}
 ((1, 1), 1, (false, false), ())) => 1×1×1×1 StridedViews.StridedView{ComplexF64,
 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 -0.9999999999999997 + 0.0im

 * (FusionTree{Irrep[SU₂]}((1, 1), 2, (false, false), ()), FusionTree{Irrep[SU₂]}
 ((1, 1), 2, (false, false), ())) => 1×1×1×1 StridedViews.StridedView{ComplexF64,
 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 1.0 + 0.0im
```

We demonstrated this entire procedure of extracting the reduced tensor elements of a symmetric tensor map for each fusion tree by projecting out the corresponding fusion tensors as an explicit illustration of how symmetric tensor maps work under the hood. In practice however, there is no need to perform this procedure explicitly. Given a dense array representing the matrix elements of a tensor map in the irrep basis, we can convert this to the corresponding symmetric tensor map by passing the data array to the `TensorMap` constructor along with the corresponding spaces,



```julia
SS_auto = TensorMap(SS_arr, V ⊗ V ← V ⊗ V)
@test SS_auto ≈ SS
```



```
Test Passed
```

Warning

While the example demonstrated here seems fairly straightforward, there's some inherent challenges to directly initializing a symmetric tensor map from a full dense array. A first important point to reiterate here is that in order for this procedure to work, we had to initialize `SS_arr` by assuming an internal basis convention for the SU_2 representation space $V^{(1)}$ that is consistent with the convention used by `fusiontensor`. While that choice here, corresponding to an ascending magnetic quantum number $m = -1, 0, 1$, seems quite natural, for many symmetries there is

no transparent natural choice. In those cases, the only way to use this approach is to explicitly check the basis convention used by `TensorKitSectors.fusiontensor` for that specific symmetry. On top of this, there are some additional complications when considering graded spaces which contain multiple sectors with non-trivial degeneracies. In that case, to even initialize the dense data array in the first place, you would need to know the order in which the sectors appear in each space internally. This information can be obtained by calling `axes(V, c)` , where `V` and `c` are either an `ElementarySpace` and a `Sector` , or a `ProductSpace` and a `Tuple` of `Sector` s respectively.

An 'elegant' approach to the Heisenberg model

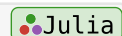
As noted above, the explicit procedure of projecting out the reduced tensor elements from the action of an operator in the irrep basis can be a bit cumbersome for more complicated groups. However, using some basic representation theory we can bypass this step altogether for the Heisenberg model. First, we rewrite the exchange interaction in the following way:

$$\vec{S}_i \cdot \vec{S}_j = \frac{1}{2}\left(\left(\vec{S}_i + \vec{S}_j\right)^2 - \vec{S}_i^2 - \vec{S}_j^2\right)$$

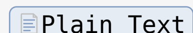
Here, $\vec{S}_i$ and $\vec{S}_j$ are spin operators on the physical irrep, while total spin operator $\vec{S}_i + \vec{S}_j$ can be decomposed onto the different coupled irreps k. It is a well known fact that the quadratic sum of the generators of SU_2, often refered to as the *quadratic Casimir*, commutes with all generators. By Schur's lemma, it must then act proportionally to the identity on every irrep, where the corresponding eigenvalue is determined by the spin irrep label. In particular, we have for each irrep l

$$\vec{S}^2 \; |l, m\rangle = l(l+1) \; |l, m\rangle.$$

It then follows from Eq. \eqref{eq:casimir*decomp} that the reduced tensor elements of the exchange interaction are completely determined by the eigenvalue of the quadratic Casimir on the uncoupled and coupled irreps. Indeed, to each fusion tree we can associate a well-defined value ![SU2*fusiontrees](img/
symmetric*tutorial/SU2*fusiontrees.svg) This gives us all we need to directly construct the exchange interaction as a symmetric `TensorMap` ,



```
V = SU2Space(1 => 1)
SS = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
for (s, f) in fusiontrees(SS)
    l3 = f.uncoupled[1].j
    l4 = f.uncoupled[2].j
    k = f.coupled.j
    SS[s, f] .= (k * (k + 1) - l3 * (l3 + 1) - l4 * (l4 + 1)) / 2
end
subblocks(SS)
```



```
subblocks(::TensorMap{ComplexF64, Rep[SU₂], 2, 2,
Vector{ComplexF64}}):
 * (FusionTree{Irrep[SU₂]}((1, 1), 0, (false, false), ()), FusionTree{Irrep[SU₂]}
 ((1, 1), 0, (false, false), ())) => 1×1×1×1 StridedViews.StridedView{ComplexF64,
 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 -2.0 + 0.0im
```

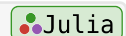
```
 * (FusionTree{Irrep[SU₂]}((1, 1), 1, (false, false), ()), FusionTree{Irrep[SU₂]}
 ((1, 1), 1, (false, false), ())) => 1×1×1×1 StridedViews.StridedView{ComplexF64,
 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 -1.0 + 0.0im

 * (FusionTree{Irrep[SU₂]}((1, 1), 2, (false, false), ()), FusionTree{Irrep[SU₂]}
 ((1, 1), 2, (false, false), ())) => 1×1×1×1 StridedViews.StridedView{ComplexF64,
 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 1.0 + 0.0im
```

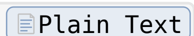
which gives exactly the same result as the previous approach.

> **Note**
>
> This last construction for the exchange interaction immediately generalizes to any value of the physical spin. All we need is to fill in the appropriate values for the uncoupled irreps l_1, l_2, l_3 and l_4.

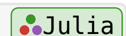
SU_N generalization

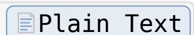
We end this subsection with some comments on the generalization of the above discussion to SU_N. As foreshadowed above, the irreps of SU_N in general have an even more complicated structure. In particular, they can admit so-called *fusion multiplicities*, where the fusion of two irreps can have not only multiple distinct outcomes, but they can even fuse to a given irrep in multiple inequivalent ways. We can demonstrate this behavior for the adjoint representation of SU_3. For this we can use the the SUNRepresentations.jl package which provides an interface for working with irreps of SU_N and their Clebsch-Gordan coefficients. A particular representation is represented by an `SUNIrrep{N}` which can be used with TensorKit.jl. The eight-dimensional adjoint representation of SU_3 is given by

```julia
l = SU3Irrep("8")
```


```
Irrep[SU₃]("8")
```


If we look at the possible outcomes of fusing two adjoint irreps, we find the by now familiar non-Abelian fusion behavior,

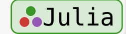
```julia
collect(l ⊗ l)
```


```
5-element Vector{SUNRepresentations.SU3Irrep}:
 "1"
 "27"
 "10"
 "8"
 "10⁺"
```


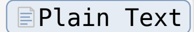
However, this particular fusion has multiplicities, since the adjoint irrep can actually fuse to itself in two distinct ways. The full decomposition of this fusion product is given by

$$\mathbf{8} \otimes \mathbf{8} = \mathbf{1} \oplus \mathbf{3} \oplus 2 \cdot \mathbf{8} \oplus \mathbf{10} \oplus \overline{\mathbf{10}} \oplus \mathbf{27}$$

This fusion multiplicity can be detected by using `Nsymbol` method from TensorKit.jl to inspect the number of times `l` appears in the fusion product `l ⊗ l`,



```julia
Nsymbol(l, l, l)
```



```
2
```

When working with irreps with fusion multiplicities, each `FusionTree` carries additional `vertices` labels which label which of the distinct fusion vertices is being referred to. We will return to this at the end of this section.

Given the generators T^k of SU_N, we can define a generalized Heisenberg model using a similar exchange interaction, giving the Hamiltonian

$$H = J \sum_{\langle i,j \rangle} \vec{T}_i \cdot \vec{T}_j$$

For a particular choice of physical irrep, the exchange interaction can again be constructed as a symmetric `TensorMap` by first rewriting it as

$$\vec{T}_i \cdot \vec{T}_j = \frac{1}{2}\left(\left(\vec{T}_i + \vec{T}_j\right)^2 - \vec{T}_i^2 - \vec{T}_j^2\right).$$

For any N, the quadratic Casimir

$$\Omega = \sum_k T^k T^k$$

commutes with all SU_N generators, meaning it has a well defined eigenvalue in each irrep. This observation then immediately given the reduced tensor elements of the exchange interaction as

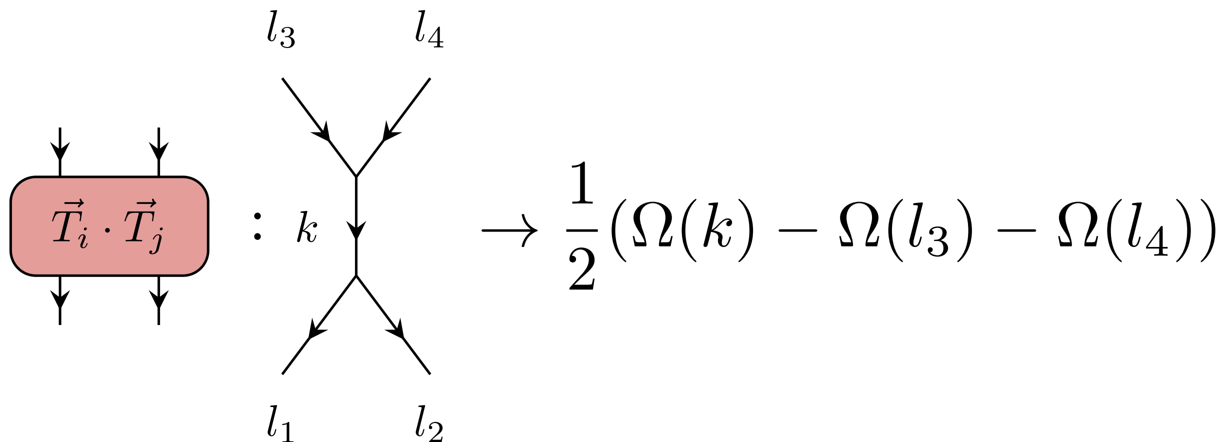


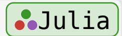
Figure 17.12: SUN_fusiontrees

Using these to directly construct the corresponding symmetric `TensorMap` is much simpler than going through the explicit projection procedure using Clebsch-Gordan coefficients.

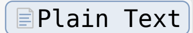
For the particular example of SU_3, the generators are given by $T^k = \frac{1}{2}\lambda^k$, where λ^k are the Gell-Mann matrices. Each irrep can be labeled as $l = D(p, q)$ where p and q are refered to as the *Dynkin labels*. The eigenvalue of the quadratic Casimir for a given irrep is given by Freudenthal's formula,

$$\Omega(D(p, q)) = \frac{1}{3}(p^2 + q^2 + 3p + 3q + pq).$$

Using SUNRepresentations.jl, we can compute the Casimir as

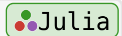


```
function casimir(l::SU3Irrep)
    p, q = dynkin_label(l)
    return (p^2 + q^2 + 3 * p + 3 * q + p * q) / 3
end
```

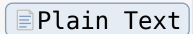


```
casimir (generic function with 1 method)
```

If we use the adjoint representation of SU_3 as physical space, the Heisenberg exchange interaction can then be constructed as



```
V = Vect[SUNIrrep{3}](SU3Irrep("8") => 1)
TT = zeros(ComplexF64, V ⊗ V ← V ⊗ V)
for (s, f) in fusiontrees(TT)
    l3 = f.uncoupled[1]
    l4 = f.uncoupled[2]
    k = f.coupled
    TT[s, f] .= (casimir(k) - casimir(l3) - casimir(l4)) / 2
end
subblocks(TT)
```

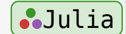


```
subblocks(::TensorMap{ComplexF64, Rep[SU₃], 2, 2,
Vector{ComplexF64}}):
 * (FusionTree{Irrep[SU₃]}(("8", "8"), "1", (false, false), (), (1,)),
 FusionTree{Irrep[SU₃]}(("8", "8"), "1", (false, false), (), (1,))) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 -3.0 + 0.0im

 * (FusionTree{Irrep[SU₃]}(("8", "8"), "8", (false, false), (), (1,)),
 FusionTree{Irrep[SU₃]}(("8", "8"), "8", (false, false), (), (1,))) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 -1.5 + 0.0im

 * (FusionTree{Irrep[SU₃]}(("8", "8"), "8", (false, false), (), (2,)),
 FusionTree{Irrep[SU₃]}(("8", "8"), "8", (false, false), (), (1,))) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 -1.5 + 0.0im

 * (FusionTree{Irrep[SU₃]}(("8", "8"), "8", (false, false), (), (1,)),
 FusionTree{Irrep[SU₃]}(("8", "8"), "8", (false, false), (), (2,))) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 -1.5 + 0.0im
```

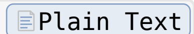
```
 * (FusionTree{Irrep[SU₃]}(("8", "8"), "8", (false, false), (), (2,)),
 FusionTree{Irrep[SU₃]}(("8", "8"), "8", (false, false), (), (2,))) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 -1.5 + 0.0im

 * (FusionTree{Irrep[SU₃]}(("8", "8"), "10", (false, false), (), (1,)),
 FusionTree{Irrep[SU₃]}(("8", "8"), "10", (false, false), (), (1,))) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FusionTree{Irrep[SU₃]}(("8", "8"), "10⁺", (false, false), (), (1,)),
 FusionTree{Irrep[SU₃]}(("8", "8"), "10⁺", (false, false), (), (1,))) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0 + 0.0im

 * (FusionTree{Irrep[SU₃]}(("8", "8"), "27", (false, false), (), (1,)),
 FusionTree{Irrep[SU₃]}(("8", "8"), "27", (false, false), (), (1,))) => 1×1×1×1
 StridedViews.StridedView{ComplexF64, 4, Memory{ComplexF64}, typeof(identity)}:
[:, :, 1, 1] =
 1.0 + 0.0im
```

Circling back to our earlier remark, we clearly see that the fusion trees of this tensor indeed have non-trivial vertex labels.

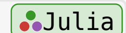
```
f = collect(fusiontrees(TT))[4][2]
```



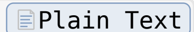
```
FusionTree{Irrep[SU₃]}(("8", "8"), "8", (false, false), (), (2,))
```



```
f.vertices
```



```
(2,)
```



Note

While we have given an explicit example using SU_3 with the adoint irrep on the physical level, the same construction holds for the general SU_N with arbitrary physical irreps. All we require is the expression for the eigenvalues of the quadratic Casimir in each irrep.

17.6 Level 5: Anyonic Symmetries and the Golden Chain

While we have focussed exclusively on group-like symmetries in our discussion so far, the framework of symmetric tensors actually extends beyond groups to so-called *categorical symmetries*. These are quite exotic symmetries characterized in terms of the topological data of a unitary fusion category. While the precise details of all the terms in these statements fall beyond the scope of this tutorial, we can give a simple example of a Hamiltonian model with a categorical symmetry called the golden chain.

This is a one-dimensional system defined as a spin chain, where each physical 'spin' corresponds to a so-called Fibonacci anyon. There are two such Fibonacci anyons, which we will denote as 1 and τ. They obey the fusion rules

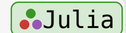
$$1 \otimes 1 = 1, \quad 1 \otimes \tau = \tau, \quad \tau \otimes \tau = 1 \oplus \tau.$$

The Hilbert space of a chain of Fibonacci anyons is not a regular tensor product space, but rather a *constrained Hilbert space* where the only allowed basis states are labeled by valid Fibonacci fusion configurations. In the golden chain model, we define a nearest-neighbor Hamiltonian on this Hilbert space by imposing an energy penalty when two neighboring anyons fuse to a τ anyon.

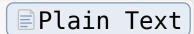
Even just writing down an explicit expression for this interaction on such a constrained Hilbert space is not entirely straightforward. However, using the framework of symmetric tensors it can actually be explicitly constructed in a very straightforward way. Indeed, TensorKit.jl supports a dedicated `FibonacciAnyon` sector type which can be used to construct precisely such a constrained Fibonacci-graded vector space. A Hamiltonian

$$H = \sum_{\langle i,j \rangle} h_{ij}$$

which favors neighboring anyons fusing to the vacuum can be constructed as a `TensorMap` on the product space of two Fibonacci-graded physical spaces



```julia
V = Vect[FibonacciAnyon](:τ => 1)
```



```
Vect[FibonacciAnyon](…) of dim 1.618033988749895:
 :τ => 1
```

and assigning the following nonzero subblock value to the two-site fusion trees

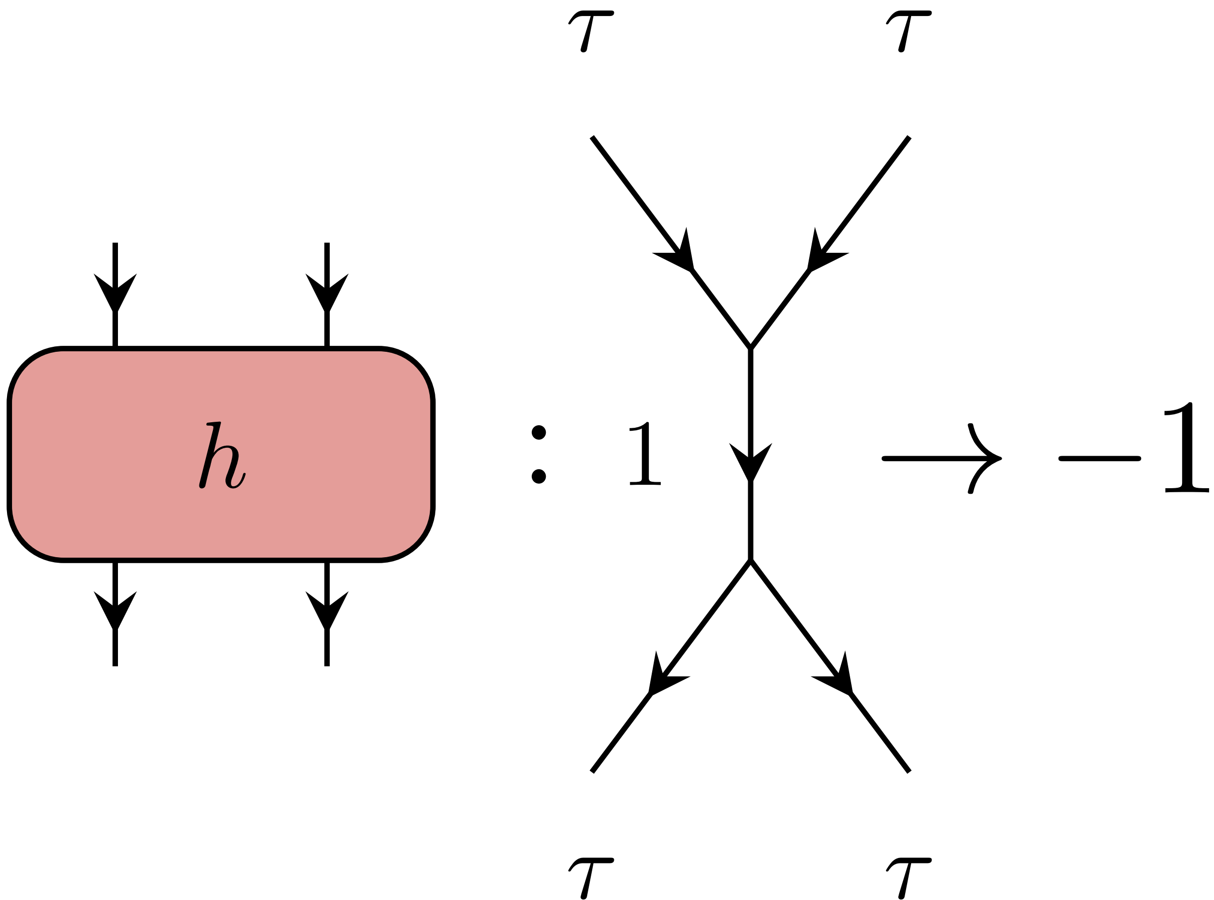


Figure 17.13: Fib_fusiontrees

This allows us to define this, at first sight, exotic and complicated Hamiltonian in a few simple lines of code,

```julia
h = ones(V ⊗ V ← V ⊗ V)
for (s, f) in fusiontrees(h)
    h[s, f] .= f.coupled == FibonacciAnyon(:I) ? -1 : 0
end
subblocks(h)
```

```
subblocks(::TensorMap{Float64, Vect[FibonacciAnyon], 2, 2,
Vector{Float64}}):
 * (FusionTree{FibonacciAnyon}((:τ, :τ), :I, (false, false), ()),
 FusionTree{FibonacciAnyon}((:τ, :τ), :I, (false, false), ())) => 1×1×1×1
 StridedViews.StridedView{Float64, 4, Memory{Float64}, typeof(identity)}:
[:, :, 1, 1] =
 -1.0


 * (FusionTree{FibonacciAnyon}((:τ, :τ), :τ, (false, false), ()),
 FusionTree{FibonacciAnyon}((:τ, :τ), :τ, (false, false), ())) => 1×1×1×1
 StridedViews.StridedView{Float64, 4, Memory{Float64}, typeof(identity)}:
[:, :, 1, 1] =
 0.0
```

Note

In the previous section we have stressed the role of Clebsch-Gordan coefficients in the structure of symmetric tensors, and how they can be used to map between the representation of an operator in the irrep basis and its symmetric tensor representation. However, for categorical symmetries such as the Fibonacci anyons, there are no Clebsch-Gordan coefficients. Therefore, the 'matrix elements of the operator in the irrep basis' are not well-defined, meaning that a Fibonacci-symmetric tensor cannot actually be converted to a plain array in a straightforward way.

Chapter 18

Optional introduction to category theory

The purpose of this page (which can safely be skipped), is to explain how certain concepts and terminology from the theory of monoidal categories apply in the context of tensors. In particular, we are interested in the category $\mathbf{Vect}$, but our concept of tensors can be extended to morphisms of any category that shares similar properties. These properties are reviewed below.

In particular, we will as example also study the more general case of $\mathbf{SVect}$, i.e. the category of super vector spaces, which contains $\mathbf{Vect}$ as a subcategory and which are useful to describe fermions.

In the end, the goal of identifying tensor manipulations in TensorKit.jl with concepts from category theory is to put the diagrammatic formulation of tensor networks in the most general context on a firmer footing. The following exposition is mostly based on[1], combined with input from[2],[3],[4], and nLab, to which we refer for further information. Furthermore, we recommend the nice introduction of[5].

18.1 Categories, functors and natural transformations

To start, a **category** C consists of

- a class $\mathrm{Ob}(C)$ of objects V, W, …
- for each pair of objects V and W, a set $\mathrm{Hom}_C(W, V)$ of morphisms $f : W \to V$ for a given map f, W is called the *domain* or *source*, and V the *codomain* or *target*.

1

```
Turaev, V. G., & Virelizier, A. (2017). Monoidal categories and
topological field theory (Vol. 322).  Birkhäuser.
```
Plain Text

2

```
Selinger, P. (2010). A survey of graphical languages for monoidal
categories.  In New structures for physics (pp. 289-355). Springer,
Berlin, Heidelberg.  [https://arxiv.org/abs/0908.3347](https://
arxiv.org/abs/0908.3347)
```
Plain Text

3

```
Kassel, C. (2012). Quantum groups (Vol. 155).  Springer Science &
Business Media.
```
Plain Text

4

```
Kitaev, A. (2006). Anyons in an exactly solved model and beyond.
Annals of Physics, 321(1), 2-111.
```
Plain Text

5

```
From categories to anyons: a travelogue Kerstin Beer, Dmytro
Bondarenko, Alexander Hahn, Maria Kalabakov, Nicole Knust, Laura
Niermann, Tobias J. Osborne, Christin Schridde, Stefan
Seckmeyer, Deniz E. Stiegemann, and Ramona Wolf [https://arxiv.
org/abs/1811.06670](https://arxiv.org/abs/1811.06670)
```
Plain Text

- composition of morphisms $f : W \to V$ and $g : X \to W$ into $(f \circ g) : X \to V$ that is associative, such that for $h : Y \to X$ we have $f \circ (g \circ h) = (f \circ g) \circ h$
- for each object V, an identity morphism $\mathrm{id}_V : V \to V$ such that $f \circ \mathrm{id}_W = f = \mathrm{id}_V \circ f$.

The morphisms in $\mathrm{Hom}_C(V, V)$ are known as endomorphism and this set is also denoted as $End_C(V)$. When the category C is clear, we can drop the subscript in $\mathrm{Hom}(W, V)$. A morphism $f : W \to V$ is an isomorphism if there exists a morphism $f^{-1} : V \to W$ called its inverse, such that $f^{-1} \circ f = \mathrm{id}_W$ and $f \circ f^{-1} = \mathrm{id}_V$.

Throughout this manual, we associate a graphical representation to morphisms and compositions thereof, which is sometimes referred to as the Penrose graphical calculus. To morphisms, we associate boxes with an incoming and outgoing line denoting the object in its source and target. The flow from source to target, and thus the direction of morphism composition $f \circ g$ (sometimes known as the flow of time) can be chosen left to right (like the arrow in $f : W \to V$), right to left (like the composition order $f \circ g$, or the matrix product), bottom to top (quantum field theory convention) or top to bottom (quantum circuit convention). Throughout this manual, we stick to this latter convention (which is not very common in manuscripts on category theory):

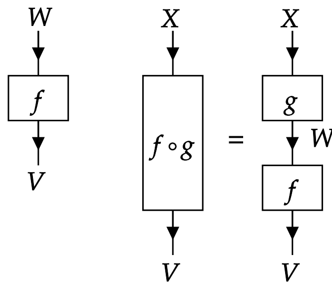


Figure 18.1: composition

The direction of the arrows, which become important once we introduce duals, are also subject to convention, and are here chosen to follow the arrow in $f : W \to V$, i.e. the source comes in and the target goes out. Strangely enough, this is opposite to the most common convention.

In the case of interest, i.e. the category $(\mathbf{Fin})\mathbf{Vect}_{\Bbbk}$ (or some subcategory thereof), the objects are (finite-dimensional) vector spaces over a field $\Bbbk$, and the morphisms are linear maps between these vector spaces with “matrix multiplication” as composition. More importantly, the morphism spaces $\mathrm{Hom}(W, V)$ are themselves vector spaces. More general categories where the morphism spaces are vector spaces over a field $\Bbbk$ (or modules over a ring $\Bbbk$) and the composition of morphisms is a bilinear operation are called $\Bbbk$-linear categories (or $\Bbbk$-algebroids, or $\mathbf{Vect}_{\Bbbk}$-enriched categories). In that case, the endomorphisms $\mathrm{End}(V)$ are a $\Bbbk$-algebra with id_V as the identity.

We also introduce some definitions which will be useful further on. A **functor** F between two categories C and D is, colloquially speaking, a mapping between categories that preserves morphism composition and identities. More specifically, $F : C \to D$ assigns to every object $V \in \mathrm{Ob}(C)$ an object $F(V) \in \mathrm{Ob}(D)$, and to each morphism $f \in \mathrm{Hom}_C(W, V)$ a morphism $F(f) \in \mathrm{Hom}_D(F(W), F(V))$ such that $F(f) \circ_D F(g) = F(f \circ_C g)$ and $F(\mathrm{id}_V) = \mathrm{id}_{F(V)}$ (where we denoted the possibly different composition laws in C and D explicitly with a subscript). In particular, every category C has an identity functor 1_C that acts trivially on objects and morphisms. Functors can also be composed. A $\Bbbk$-linear functor between two $\Bbbk$-linear categories has a linear action on morphisms.

Given two categories C and D, and two functors F and G that map from C to D, a **natural transformation** $\varphi : F \longrightarrow G$ is a family of morphisms $\varphi_V \in \mathrm{Hom}_D(F(V), G(V))$ in D, labeled by the objects

V of C, such that $\varphi_V \circ F(f) = G(f) \circ \varphi_W$ for all morphisms $f \in \mathrm{Hom}_C(W, V)$. If all morphisms φ_V are isomorphisms, φ is called a natural isomorphism and the two functors F and G are said to be *isomorphic*.

The *product* of two categories C and C', denoted $C \times C'$, is the category with objects $\mathrm{Ob}(C \times C') = \mathrm{Ob}(C) \times \mathrm{Ob}(C')$, whose elements are denoted as tuples (V, V'), and morphisms $\mathrm{Hom}_{C\times C'}((W, W'), (V, V')) = \mathrm{Hom}_C(W, V) \times \mathrm{Hom}_{C'}(W', V')$. Composition acts as $(f, f') \circ (g, g') = (f \circ f', g \circ g')$ and the identity is given by $\mathrm{id}_{V,V'} = (\mathrm{id}_V, \mathrm{id}_{V'})$. In a similar fashion, we can define the *product of functors* $F : C \to D$ and $F' : C' \to D'$ as a functor $F \times F' : (C \times C') \to (D \times D')$ mapping objects (V, V') to $(F(V), F'(V'))$ and morphisms (f, f') to $(F(f), F'(f'))$.

18.2 Monoidal categories

The next property of the category $\mathbf{Vect}$ that we want to highlight and generalize is that which allows to take tensor products. Indeed, a category C is said to be a **tensor category** (a.k.a. a *monoidal category*), if it has

- a binary operation on objects $\otimes : \mathrm{Ob}(C) \times \mathrm{Ob}(C) \to \mathrm{Ob}(C)$
- a binary operation on morphisms, also denoted as $\otimes$, such that $\otimes : \mathrm{Hom}_C(W_1, V_1) \times \mathrm{Hom}_C(W_2, V_2) \to \mathrm{Hom}_C(W_1 \otimes W_2, V_1 \otimes V_2)$
- an identity or unit object I
- three families of natural isomorphisms:
  - $\forall V \in \mathrm{Ob}(C)$, a left unitor (a.k.a. left unitality constraint) $\lambda_V : I \otimes V \to V$
  - $\forall V \in \mathrm{Ob}(C)$, a right unitor (a.k.a. right unitality constraint) $\rho_V : V \otimes I \to V$
  - $\forall V_1, V_2, V_3 \in \mathrm{Ob}(C)$, an associator (a.k.a. associativity constraint) $\alpha_{V_1,V_2,V_3} : (V_1 \otimes V_2) \otimes V_3 \to V_1 \otimes (V_2 \otimes V_3)$ that satisfy certain consistency conditions (coherence axioms), which are known as the *pentagon equation* (stating that the two possible mappings from $(((V_1 \otimes V_2) \otimes V_3) \otimes V_4)$ to $(V_1 \otimes (V_2 \otimes (V_3 \otimes V_4)))$ are compatible) and the *triangle equation* (expressing compatibility between the two possible ways to map $((V_1 \otimes I) \otimes V_2)$ to $(V_1 \otimes (I \otimes V_2)))$.

In terms of functors and natural transformations, $\otimes$ is a functor from the product category $C \times C$ to C. Furthermore, the left (or right) unitor λ (or ρ) is a natural isomorphism between a nameless functor $C \to C$ that maps objects $V \to I \otimes V$ (or $V \to V \otimes I$) and the identity functor 1_C. Similarly, the associator α is a natural isomorphism between the two functors $\otimes (\otimes \times 1_C)$ and $\otimes (1_C \times \otimes)$ from $C \times C \times C$ to C. In a k-linear category, the tensor product of morphisms is also a bilinear operation. A monoidal category is said to be *strict* if $I \otimes V = V = V \otimes I$ and $(V_1 \otimes V_2) \otimes V_3 = V_1 \otimes (V_2 \otimes V_3)$, and the left and right unitor and associator are just the identity morphisms for these objects.

For the category $\mathbf{Vect}$, the identity object I is just the scalar field $\Bbbk$ over which the vector spaces are defined, and which can be identified with a one-dimensional vector space. This is not automatically a strict category, especially if one considers how to represent tensor maps on a computer. The distinction between V, $I \otimes V$ and $V \otimes I$ amounts to adding or removing an extra factor I to the tensor product structure of the domain or codomain, and so the left and right unitor are analogous to removing extra dimensions of size 1 from a multidimensional array. The fact that arrays with and without additional dimensions 1 are not automatically identical and an actual operation is required to insert or remove them, has led to some discussion in several programming languages that provide native support for multidimensional arrays.

For what concerns the associator, the distinction between $(V_1 \otimes V_2) \otimes V_3$ and $V_1 \otimes (V_2 \otimes V_3)$ is typically absent for simple tensors or multidimensional arrays. However, this grouping can be taken to indicate how to build the fusion tree for coupling irreps to a joint irrep in the case of symmetric tensors. As such, going from one to the other requires a recoupling (F-move) which has a non-trivial action on the reduced blocks. We elaborate on this in the context of Fusion categories below. However, we can already note that we will always represent tensor products using a canonical order $(\ldots((V_1 \otimes V_2) \otimes V_3)\ldots \otimes V_N)$. A similar approach can be followed to turn any tensor category into a strict tensor category (see Section XI.5 of[6]).

The different natural isomorphisms involving the unit object have various relations, such as $\lambda_{V\otimes W} \circ \alpha_{I,V,W} = \lambda_V \otimes \mathrm{id}_W$ and $\lambda_I = \rho_I : I \otimes I \to I$. The last relation defines an isomorphism between $I \otimes I$ and I, which can also be used to state that for $f, g \in End_C(I)$, $f \circ g = \rho_I \circ (f \otimes g) \circ \lambda_I^{-1} = g \circ f$. Hence, the tensor product of morphisms in $End_C(I)$ can be related to morphism composition in $End_C(I)$, and furthermore, the monoid of endomorphisms $End_C(I)$ is commutative (abelian). In the case of a $\Bbbk$-linear category, it is an abelian $\Bbbk$-algebra. In the case of $\mathbf{Vect}$, $\mathrm{End}(I)$ is indeed isomorphic to the field of scalars $\Bbbk$. We return to the general case where $End_C(I)$ is isomorphic to $\Bbbk$ itself in the section on pre-fusion categories.

Furthermore, *Mac Lane's coherence theorem* states that the triangle and pentagon condition are sufficient to ensure that any consistent diagram made of associators and left and right unitors (involving all possible objects in C) commutes. For what concerns the graphical notation, the natural isomorphisms will not be represented and we make no distinction between $(V_1 \otimes V_2) \otimes V_3$ and $V_1 \otimes (V_2 \otimes V_3)$. Similarly, the identity object I can be added or removed at will, and when drawn, is often represented by a dotted or dashed line. Note that any consistent way of inserting the associator or left or right unitor to convert a graphical representation to a diagram of compositions and tensor products of morphisms gives rise to the same result, by virtue of Mac Lane's coherence theorem. Using the horizontal direction (left to right) to stack tensor products, this gives rise to the following graphical notation for the tensor product of two morphisms, and for a general morphism t between a tensor product of objects in source and target:

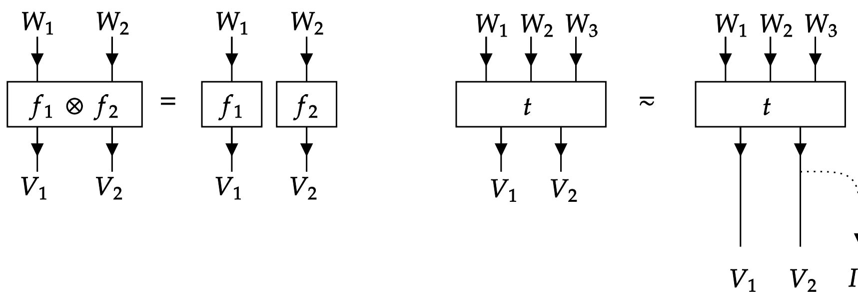


Figure 18.2: tensorproduct

Another relevant example is the category $\mathbf{SVect}_{\Bbbk}$, which has as objects *super vector spaces* over $\Bbbk$, which are vector spaces with a $\mathbb{Z}_2$ grading, i.e. they are decomposed as a direct sum $V = V_0 \oplus V_1$. Furthermore, the morphisms between two super vector spaces are restricted to be grading preserving, i.e. $f \in \mathrm{Hom}_{\mathbf{SVect}}(W, V)$ has $f(W_0) \subset V_0$ and $f(W_1) \subset V_1$. The graded tensor product between two

[6]

```
Kassel, C. (2012). Quantum groups (Vol. 155).  Springer Science & Business Media.
```
Plain Text

super vector spaces is defined as $(V \otimes_{\mathrm{g}} W) = (V \otimes_{\mathrm{g}} W)_0 \oplus (V \otimes_{\mathrm{g}} W)_1$ with $(V \otimes_{\mathrm{g}} W)_0 = (V_0 \otimes W_0) \oplus (V_1 \otimes W_1)$ and $(V \otimes_{\mathrm{g}} W)_1 = (V_0 \otimes W_1) \oplus (V_1 \otimes W_0)$. The unit object I is again isomorphic to $\Bbbk$, i.e. $I_0 = \Bbbk$ and $I_1 = 0$, a zero-dimensional vector space. In particular, the category $\mathbf{SVect}_{\Bbbk}$ contains $\mathbf{Vect}_{\Bbbk}$ as a (monoidal) subcategory, by only selecting those objects V for which $V_1 = 0$. We will return to the example of $\mathbf{SVect}$ throughout the remainder of this page.

Finally, we generalize the notion of a functor between monoidal categories. A *monoidal functor* between two tensor categories $(C, \otimes_C, I_C, \alpha_C, \lambda_C, \rho_C)$ and $(D, \otimes_D, I_D, \alpha_D, \lambda_D, \rho_D)$ is a functor $F : C \to D$ together with two monoidal constraints, namely

- a morphism $F_{\ 0} : I_D \to F(I_C)$
- a natural transformation $F_2 = F_2(X, Y) : F(X) \otimes_D F(Y) \to F(X \otimes_C Y), \forall X, Y \in \mathrm{Ob}(C)$ between the functors $\otimes_D (F \times F)$ and $F \otimes_C$ from $C \times C$ to D. A *monoidal natural transformation* φ between two monoidal functors $F : C \to D$ and $G : C \to D$is a natural transformation $\varphi : F \longrightarrow G$ that furthermore satisfies
  - $\varphi_{I_C} F_0 = G_0$
  - $\forall X, Y \in \mathrm{Ob}(C)$: $\varphi_{X \otimes Y} F_2(X, Y) = G_2(X, Y)(\varphi_X \otimes \varphi_Y)$.

For further reference, we also define the following categories which can be associated with the category $\mathcal{C} = (C, \otimes, I, \alpha, \lambda, \rho)$

- $\mathcal{C}^{\mathrm{op}} = (C^{\mathrm{op}}, \otimes, I, \alpha^{\mathrm{op}}, \lambda^{\mathrm{op}}, \rho^{\mathrm{op}})$ where the opposite category C^{op} has the same objects as C but has $\mathrm{Hom}_{C^{\mathrm{op}}}(X, Y) = \mathrm{Hom}_C(Y, X)$ and a composition law $g \circ^{\mathrm{op}} f = f \circ g$, with $\circ$ the composition law of C. Furthermore, we have $\alpha^{\mathrm{op}}_{X,Y,Z} = (\alpha_{X,Y,Z})^{-1}$, $\lambda^{\mathrm{op}}_X = (\lambda_X)^{-1}$ and $\rho^{\mathrm{op}}_X = (\rho_X)^{-1}$
- $\mathcal{C}^{\otimes\mathrm{op}} = (C, \otimes^{\mathrm{op}}, I, \alpha^{\otimes\mathrm{op}}, \lambda^{\otimes\mathrm{op}}, \rho^{\otimes\mathrm{op}})$ where the functor $\otimes^{\mathrm{op}} : C \times C \to C$ is the opposite monoidal product, which acts as $X \otimes^{\mathrm{op}} Y = Y \otimes X$ on objects and similar on morphisms. Furthermore, $\alpha^{\otimes\mathrm{op}}_{X,Y,Z} = (\alpha_{Z,Y,X})^{-1}$, $\lambda^{\otimes\mathrm{op}}_X = \rho_X$ and $\rho^{\otimes\mathrm{op}}_X = \lambda_X$
- The two previous transformations (which commute) composed: $\mathcal{C}^{\mathrm{rev}} = (C^{\mathrm{op}}, \otimes^{\mathrm{op}}, I, \alpha^{\mathrm{rev}}, \lambda^{\mathrm{rev}}, \rho^{\mathrm{rev}})$ with $\alpha^{\mathrm{rev}}_{X,Y,Z} = \alpha_{Z,Y,X}$, $\lambda^{\mathrm{rev}}_X = (\rho_X)^{-1}$, $\rho^{\mathrm{rev}}_X = (\lambda_X)^{-1}$.

18.3 Duality: rigid, pivotal and spherical categories

Another property of the category $\mathbf{Vect}$ that we want to generalize is the notion of duals. For a vector space V, i.e. an object of $\mathbf{Vect}$, the dual V^* is itself a vector space. Evaluating the action of dual vector on a vector can, because of linearity, be interpreted as a morphism from $V^* \otimes V$ to I. Note that elements of a vector space V have no categorical counterpart in themselves, but can be interpreted as morphism from I to V. To map morphisms from $\mathrm{Hom}(W, V)$ to elements of $V \otimes W^*$, i.e. morphisms in $\mathrm{Hom}(I, V \otimes W^*)$, we use another morphism $\mathrm{Hom}(I, W \otimes W^*)$ which can be considered as the inverse of the evaluation map.

Hence, duality in a monoidal category is defined via an *exact pairing*, i.e. two families of non-degenerate morphisms, the evaluation (or co-unit) $\epsilon_V : {}^\vee V \otimes V \to I$ and the coevaluation (or unit) $\eta_V : I \to V \otimes {}^\vee V$ which satisfy the “snake rules”:

$$\rho_V \circ (\mathrm{id}_V \otimes \epsilon_V) \circ (\eta_V \otimes \mathrm{id}_V) \circ \lambda_V^{-1} = \mathrm{id}_V$$
$$\lambda^{-1}_{{}^\vee V} \circ (\epsilon_V \otimes \mathrm{id}_{{}^\vee V}) \circ (\mathrm{id}_{{}^\vee V} \otimes \eta_V) \circ \rho^{-1}_{{}^\vee V} = \mathrm{id}_{{}^\vee V}$$

and can be used to define an isomorphism between $\mathrm{Hom}(W \otimes V, U)$ and $\mathrm{Hom}(W, U \otimes {}^\vee V)$ for any triple of objects $U, V, W \in \mathrm{Ob}(C)$. Note that if there are different duals (with corresponding exact

pairings) associated to an object V, a mixed snake composition using the evaluation of one and coevaluation of the other duality can be used to construct an isomorphism between the two associated dual objects. Hence, duality is unique up to isomorphisms.

For (real or complex) vector spaces, we denote the dual as V^*, a notation that we preserve for pivotal categories (see below). Using a bra-ket notation and a generic basis $\mid n\rangle$ for V and dual basis $\langle m \mid$ for V^* (such that $\langle m \mid n\rangle = \delta_{m,n}$), the evaluation is given by $\epsilon_V : {}^\vee V \otimes V \to \mathbb{C} : \langle m \mid \otimes \mid n\rangle \mapsto \delta_{m,n}$ and the coevaluation or unit is $\eta_V : \mathbb{C} \to V \otimes {}^\vee V : \alpha \mapsto \alpha \sum_n \mid n\rangle \otimes \langle n \mid$. Note that this does not require an inner product, i.e. no relation or mapping from $\mid n\rangle$ to $\langle n \mid$ was defined. For a general tensor map $t : W_1 \otimes W_2 \otimes \ldots \otimes W_{N_2} \to V_1 \otimes V_2 \otimes \ldots \otimes V_{N_1}$, by successively applying $\eta_{W_{N_2}}, \eta_{W_{N_2-1}}, \ldots, \eta_{W_1}$ (in combination with the left or right unitor), we obtain a tensor in $V_1 \otimes V_2 \otimes \ldots \otimes V_{N_1} \otimes W_{N_2}^* \otimes \ldots \otimes W_1^*$. Hence, we can define or identify $(W_1 \otimes W_2 \otimes \ldots \otimes W_{N_2})^* = W_{N_2}^* \otimes \ldots \otimes W_1^*$. Indeed, it can be shown that for any category which has duals for objects V and W, an exact pairing between $V \otimes W$ and ${}^\vee W \otimes {}^\vee V$ can be constructed out of the evaluation and coevaluation of V and W, such that ${}^\vee W \otimes {}^\vee V$ is at least isomorphic to ${}^\vee(V \otimes W)$.

Graphically, we represent the exact pairing and snake rules as

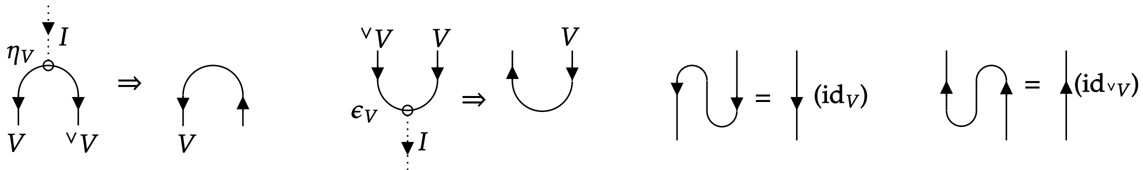


Figure 18.3: left dual

Note that we denote the dual objects ${}^\vee V$ as a line V with arrows pointing in the opposite (i.e. upward) direction. This notation is related to quantum field theory, where anti-particles are (to some extent) interpreted as particles running backwards in time.

These exact pairings are known as the left evaluation and coevaluation, and ${}^\vee V$ is the left dual of V. Likewise, we can also define a right dual $V^\vee$ of V and associated pairings, the right evaluation $\tilde{\epsilon}_V : V \otimes V^\vee \to I$ and coevaluation $\tilde{\eta}_V : I \to V^\vee \otimes V$, satisfying

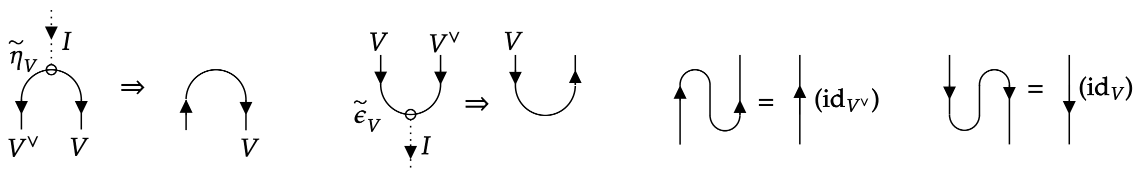


Figure 18.4: right dual

In particular, one could choose $\tilde{\epsilon}_{{}^\vee V} = \epsilon_V$ and thus define V as the right dual of ${}^\vee V$. While there might be other choices, this choice must at least be isomorphic, such that $({}^\vee V)^\vee \approx V$.

If objects V and W have left (respectively right) duals, then for a morphism $f \in \mathrm{Hom}(W, V)$, we furthermore define the left (respectively right) *transpose* ${}^\vee f \in \mathrm{Hom}({}^\vee V, {}^\vee W)$ (respectively $f^\vee \in \mathrm{Hom}(V^\vee, W^\vee)$) as

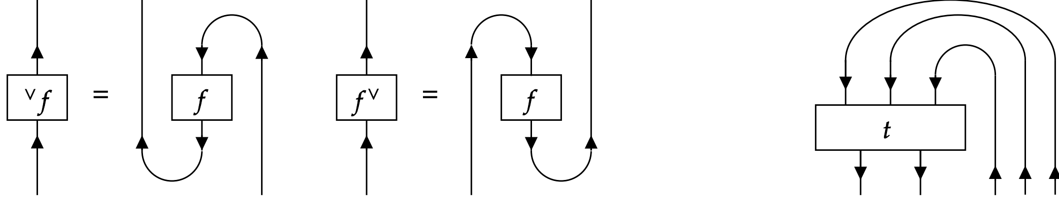


Figure 18.5: transpose

where on the right we also illustrate the mapping from $t \in \mathrm{Hom}(W_1 \otimes W_2 \otimes W_3, V_1 \otimes V_2)$ to a morphism in $\mathrm{Hom}(I, V_1 \otimes V_2 \otimes {}^{\vee}W_3 \otimes {}^{\vee}W_2 \otimes {}^{\vee}W_1)$.

Note that the graphical notation, at least the lines with opposite arrows, do not allow to distinguish between the right dual $V^{\vee}$ and the left dual ${}^{\vee}V$. We come back to this point below.

A left (or right) duality in a (monoidal) category is now defined as an association of a left (or right) dual with every object of the category, with corresponding exact pairings, and a category admitting such a duality is a left (or right) **rigid category** (or left or right autonomous category). Given that left (or right) morphism transposition satisfies ${}^{\vee}(f \circ g) = {}^{\vee}g \circ {}^{\vee}f = {}^{\vee}f \circ^{\mathrm{op}} {}^{\vee}g$ and recalling ${}^{\vee}(V \otimes W) = {}^{\vee}W \otimes {}^{\vee}V$ (and similar for right duality), we can define duality in a functorial way. A (left or right) rigid category $\mathcal{C}$ is a category which admits a (left or right) duality functor, i.e. a functor from $\mathcal{C}$ to $\mathcal{C}^{\mathrm{rev}}$ that maps objects to its (left or right) dual, and morphisms to its (left or right) transpose. In particular, the snake rules can now be read as the functioral requirement that ${}^{\vee}(\mathrm{id}_V) = \mathrm{id}_{{}^{\vee}V}$.

In all of this, left and right duality can be completely distinct. Equivalently, the left dual of the left dual of an object V, i.e. ${}^{\vee\vee}V$ is not necessarily V itself, nor do the exact pairings enable us to construct an isomorphism between ${}^{\vee\vee}V$ and V. For finite-dimensional vector spaces, however, ${}^{\vee\vee}V$ and V, or thus ${}^{\vee}V$ and $V^{\vee}$ are known to be isomorphic. The categorical generalization is that of a **pivotal category** (or sovereign category), i.e. a monoidal category with two-sided duals $X^* = {}^{\vee}X = X^{\vee} = X^*$ such that the left and right duality functor coincide, and thus also the left and right transpose of morphisms, i.e. $f^* = {}^{\vee}f = f^{\vee} \in \mathrm{Hom}(V^*, W^*)$ for any $f \in \mathrm{Hom}(W, V)$. Given that $\tilde{\epsilon}_X$ and $\tilde{\eta}_X$ can be interpreted as an exact pairing ϵ_{X^*} and η_{X^*}, this can be used to recognize X as a left dual of X^*, which is then not necessarily equal but at least isomorphic to X^{**} with the isomorphism given by the mixed snake composition alluded to in the beginning of this section, i.e. $\delta_X : X \to X^{**}$ given by $\delta_X = (\tilde{\epsilon}_X \otimes \mathrm{id}_{X^*}) \circ (\mathrm{id}_X \otimes \eta_{X^*})$. A more formal statement is that δ is a natural isomorphism between the double dual functor and the identity functor of a category C. In a similar manner, such a δ can be used to define a natural isomorphism between left and right dual functor (which is a slight generalization of the above definition of a pivotal category), and as such it is often called the *pivotal structure.*

Hence, in a pivotal category, left and right duals are the same or isomorphic, and so are objects and their double duals. As such, we will not distinguish between them in the graphical representation and suppress the natural isomorphism δ. Note, as already suggested by the graphical notation above, that we can interpret transposing a morphism as rotating its graphical notation by 180 degrees (either way).

Furthermore, in a pivotal category, we can define a map from $\mathrm{End}(V)$, the endomorphisms of an object V to endomorphisms of the identity object I, i.e. the field of scalars in the case of the category $\mathbf{Vect}$, known as the trace of f. In fact, we can define both a left trace as

$$\mathrm{tr}_{\mathrm{l}}(f) = \epsilon_V \circ (\mathrm{id}_{V^*} \otimes f) \circ \tilde{\eta}_V$$

and a right trace as

$$\mathrm{tr}_{\mathrm{r}}(f) = \tilde{\epsilon}_V \circ (f \otimes \mathrm{id}_{V^*}) \circ \eta_V$$

They are graphically represented as

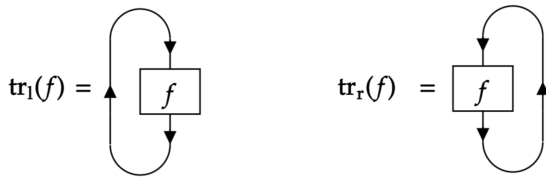


Figure 18.6: trace

and they do not need to coincide. Note that $\mathrm{tr}_{\mathrm{l}}(f) = \mathrm{tr}_{\mathrm{r}}(f^*)$ and that $\mathrm{tr}_{\mathrm{l/r}}(f \circ g) = \mathrm{tr}_{\mathrm{l/r}}(g \circ f)$. The (left or right) trace of the identity morphism id_V defines the corresponding (left or right) dimension of the object V, i.e. $\mathrm{dim}_{\mathrm{l/r}}(V) = tr_{\mathrm{l/r}}(\mathrm{id}_V)$. In a **spherical** category, both definitions of the trace coincide for all V and we simply refer to the trace $\mathrm{tr}(f)$ of an endomorphism. The particular value $\dim(V) = \mathrm{tr}(\mathrm{id}_V)$ is known as the (quantum) dimension of the object V, referred to as `dim(V)` in TensorKit.jl.

For further information and a more detailed treatment of rigid and pivotal categories, we refer to[7] and[8]. We conclude this section by studying the example of $\mathbf{SVect}$. Let us, in every super vector space V, define a basis $|\ n\rangle$ that is compatible with the grading. The value $|\ n\ | = 0, 1$ indicates that $|\ n\rangle \in V_{|\ n\ |}$. We again define a dual basis $\langle m \mid$ for V^* (such that $\langle m \mid n\rangle = \delta_{m,n}$), and then define the left evaluation by $\epsilon_V : V^* \otimes V \to \mathbb{C} : \langle m \mid \otimes_{\mathrm{g}} \mid n \rangle \to \langle m \mid n\rangle = \delta_{m,n}$ and the left coevaluation by $\eta_V : \mathbb{C} \to V \otimes V^* : \alpha \to \alpha \sum_n |\ n\rangle \otimes_{\mathrm{g}} \langle n\ |$. Note that this does not require an inner product and satisfies the snake rules. For the right evaluation and coevaluation, there are two natural choices, namely $\tilde{\epsilon}_V : V \otimes V^* \to \mathbb{C} : |\ n\rangle \otimes_{\mathrm{g}} \langle m\ | \to (\pm 1)^{|\ n\ |}\delta_{m,n}$ and $\tilde{\eta}_V : \mathbb{C} \to V^* \otimes V : \alpha \to \sum_n (\pm 1)^{|\ n\ |}\langle n \mid \otimes_{\mathrm{g}} |\ n\rangle$. The resulting trace of an endomorphism $f \in \mathrm{End}(V)$ is given by $\mathrm{tr}^{\mathrm{l}}(f) = \mathrm{tr}^{\mathrm{r}}(f) = \mathrm{tr}(f) = \sum_n (\pm 1)^{|\ n\ |}\langle n \mid f \mid n\rangle$ and is known as either the regular trace (in the case of $+1$) or the *supertrace* (in the case of -1). In particular, $\dim(V) = \dim(V_0) \pm \dim(V_1)$, and can be negative in the case of the supertrace. Both are valid choices to make $\mathbf{SVect}$ into a spherical category.

18.4 Braidings, twists and ribbons

While duality and the pivotal structure allow to move vector spaces back and forth between the domain (source) and codomain (target) of a tensor map, reordering vector spaces within the domain or codomain of a tensor map, i.e. within a tensor product $V_1 \otimes V_2 \otimes ...V_N$ requires additional structure. In particular, we need at the very least a **braided tensor category** C, which is endowed with a *braiding* τ, i.e. a natural isomorphism $\tau_{V,W} : V \otimes W \to W \otimes V_{V,W \in \mathrm{Ob}(C)}$ between the functors $\otimes$ and $\otimes^{\mathrm{op}}$ such that $\tau_{V,V'} \circ (f \otimes g) = (g \otimes f) \circ \tau_{W,W'}$ for any morphisms $f \in \mathrm{Hom}(W, V)$ and $g \in \mathrm{Hom}(W', V')$. A valid braiding needs to satisfy a coherence condition with the associator α known as the *hexagon equation*, which expresses that the braiding is $\otimes$-multiplicative, i.e. $\tau_{U,V \otimes W} = (\mathrm{id}_V \otimes \tau_{U,W})(\tau_{U,V} \otimes \mathrm{id}_W)$ and $\tau_{U \otimes V,W} = (\tau_{U,W} \otimes \mathrm{id}_V W)(\mathrm{id}_U \otimes \tau_{V,W})$ (where the associator has been omitted). We also have $\lambda_V \circ \tau_{V,I} = \rho_{V,I}$, $\rho_V \circ \tau_{I,V} = \lambda_V$ and $\tau_{V,I} = \tau_{I,V}^{-1}$ for any $V \in \mathrm{Ob}(C)$.

---

[7]

```
Turaev, V. G., & Virelizier, A. (2017). Monoidal categories and
topological field theory (Vol. 322).  Birkhäuser.
```
Plain Text

[8]

```
Selinger, P. (2010). A survey of graphical languages for monoidal
categories.  In New structures for physics (pp. 289-355). Springer,
Berlin, Heidelberg.  [https://arxiv.org/abs/0908.3347](https://
arxiv.org/abs/0908.3347)
```
Plain Text

The braiding isomorphism $\tau_{V,W}$ and its inverse are graphically represented as the lines V and W crossing over and under each other:

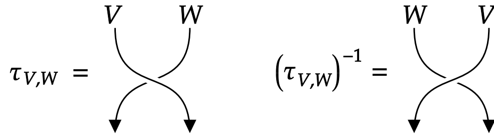


Figure 18.7: braiding

such that we have

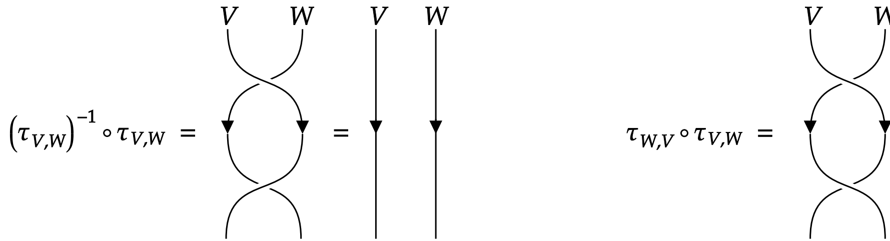


Figure 18.8: braiding relations

where the expression on the right hand side, $\tau_{W,V} \circ \tau_{V,W}$ can generically not be simplified. Hence, for general braidings, there is no unique choice to identify a tensor in $V \otimes W$ and $W \otimes V$, as the isomorphisms $\tau_{V,W}$, $\tau^{-1}_{W,V}$, $\tau_{V,W} \circ \tau_{W,V} \circ \tau_{V,W}$, ... mapping from $V \otimes W$ to $W \otimes V$ can all be different. In order for there to be a unique map from $V_1 \otimes V_2 \otimes \ldots V_N$ to any permutation of the objects in this tensor product, the braiding needs to be *symmetric*, i.e. $\tau_{V,W} = \tau^{-1}_{W,V}$ or, equivalently $\tau_{W,V} \circ \tau_{V,W} = \mathrm{id}_{V \otimes W}$. The resulting category is then referred to as a **symmetric tensor category**. In a graphical representation, it means that there is no distinction between over- and under- crossings and, as such, lines can just cross, where the crossing represents the action of $\tau_{V,W} = \tau^{-1}_{W,V}$.

In the case of the category **Vect** a valid braiding consists of just flipping the the objects/morphisms involved, e.g. for a simple cartesian tensor, permuting the tensor indices is equivalent to applying Julia's function `permutedims` on the underlying data. Less trivial braiding implementations arise in the context of tensors with symmetries (where the fusion tree needs to be reordered, as discussed in Sectors, representation spaces and fusion trees) or in the case of **SVect**, which will again be studied in detail at the end of this section.

The braiding of a space and a dual space also follows naturally, it is given by $\tau_{V^*,W} = \lambda_{W \otimes V^*} \circ (\epsilon_V \otimes \mathrm{id}_{W \otimes V^*}) \circ (\mathrm{id}_{V^*} \otimes \tau^{-1}_{V,W} \otimes \mathrm{id}_{V^*}) \circ (\mathrm{id}_{V^* \otimes W} \otimes \eta_V) \circ \rho^{-1}_{V^* \otimes W}$, i.e.

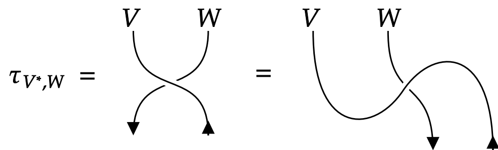


Figure 18.9: braiding dual

Balanced categories C are braided categories that come with a **twist** θ, a natural transformation from the identity functor 1_C to itself, such that $\theta_V \circ f = f \circ \theta_W$ for all morphisms $f \in \mathrm{Hom}(W, V)$, and for which the main requirement is that

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

In particular, a braided pivotal category is balanced, as we can even define two such twists, namely a left and right twist given by

$$\theta^{\mathrm{l}}_V = (\epsilon_V \otimes \mathrm{id}_V)(\mathrm{id}_{V^*} \otimes \tau_{V,V})(\tilde{\eta}_V \otimes \mathrm{id}_V)$$

and

$$\theta^{\mathrm{r}}_V = (\mathrm{id}_V \otimes \tilde{\epsilon}_V)(\tau_{V,V} \otimes \mathrm{id}_{V^*})(\mathrm{id}_V \otimes \epsilon_V)$$

where we omitted the necessary left and right unitors and associators. Graphically, the twists and their inverse (for which we refer to[9]) are then represented as

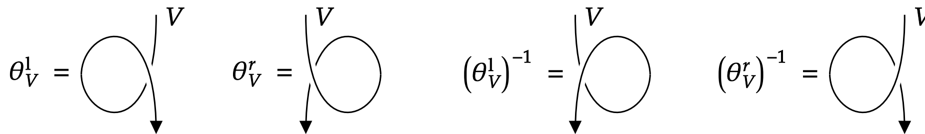


Figure 18.10: twists

The graphical representation also makes it straightforward to verify that $(\theta^{\mathrm{l}}_V)^* = \theta^{\mathrm{r}}_{V^*}$, $(\theta^{\mathrm{r}}_V)^* = \theta^{\mathrm{l}}_{V^*}$ and $\mathrm{tr}_{\mathrm{l}}(\theta^{\mathrm{r}}_V) = \mathrm{tr}_{\mathrm{r}}(\theta^{\mathrm{l}}_V)$.

When $\theta^{\mathrm{l}} = \theta^{\mathrm{r}}$, or thus, equivalently, $\theta^*_V = \theta_{V^*}$ for either θ^{l} or θ^{r}, the category is said to be **tortile** or also a **ribbon category**, because its graphical representation is compatible with the isotopy of a ribbon, i.e. where the lines representing objects are depicted as ribbons. For convenience, we continue to denote them as lines. Ribbon categories are necessarily spherical, i.e. one can prove the equivalence of the left and right trace.

Alternatively, one can start from a balanced and rigid category (e.g. with a left duality), and use the twist θ, which should satisfy $\theta^*_V = \theta_{V^*}$, to define a pivotal structure, or, to define the exact pairing for the right dual functor as

$$\tilde{\eta}_V = \tau_{V,V^*} \circ (\theta_V \otimes \mathrm{id}_{V^*}) \circ \eta_V = (\mathrm{id}_{V^*} \otimes \theta_V) \circ \tau_{V,V^*} \circ \eta_V$$

$$\tilde{\epsilon}_V = \epsilon_V \circ (\mathrm{id}_{V^*} \otimes \theta_V) \circ \tau_{V,V^*} = \epsilon_V \circ \tau_{V,V^*} \circ (\theta_V \otimes \mathrm{id}_{V^*})$$

or graphically

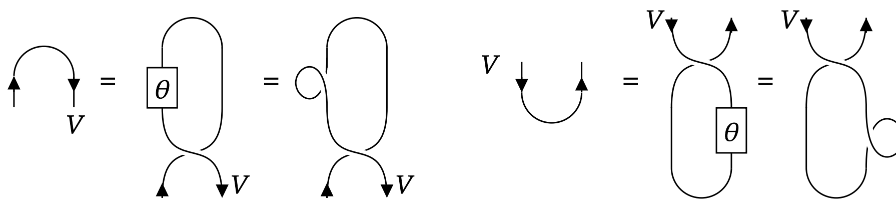


Figure 18.11: pivotal from twist

where we have drawn θ as θ^{l} on the left and as θ^{r} on the right, but in this case the starting assumption was that they are one and the same, and we defined the pivotal structure so as to make it compatible with the graphical representation. This construction of the pivotal structure can than be used to define the trace, which is spherical, i.e.

[9]

```
Turaev, V. G., & Virelizier, A. (2017). Monoidal categories and
topological field theory (Vol. 322).  Birkhäuser.
```
Plain Text

$$\mathrm{tr}(f) = \epsilon_V \circ \tau_{V,V^*} \circ ((\theta_V \circ f) \otimes \mathrm{id}_{V^*}) \circ \eta_V = \epsilon_V \circ (\mathrm{id}_{V^*} \otimes (f \circ \theta_V)) \circ \tau_{V,V^*} \circ \eta_V$$

Note finally, that a ribbon category where the braiding is symmetric, is known as a **compact closed category**. For a symmetric braiding, the trivial twist $\theta_V = \mathrm{id}_V$ is always a valid choice, but it might not be the choice that one necessarily want to use. Let us study the case of **SVect** again. Reinvoking our basis $|\, m\rangle \in V$ and $|\, n\rangle \in W$, the braiding $\tau_{V,W}$ is given by the Koszul sign rule, i.e. $\tau_{V,W} : |\, m\rangle \otimes_{\mathrm{g}} |\, n\rangle \mapsto (-1)^{|\, m\,||\, n\,|} \,|\, n\rangle \otimes_{\mathrm{g}} |\, m\rangle$. Hence, braiding amounts to flipping the two spaces, but picks up an additional minus sign if both $|\, m\rangle \in V_1$ and $|\, n\rangle \in W_1$. This braiding is symmetric, i.e. $\tau_{W,V} \circ \tau_{V,W} = \mathrm{id}_{V\otimes W}$. Between spaces and dual spaces, we similarly obtain the braiding rule $\langle m \,|\otimes_{\mathrm{g}} |\, n\rangle \mapsto (-1)^{|\, m\,||\, n\,|} \,|\, n\rangle \otimes_{\mathrm{g}} \langle m\,|$. Combining the braiding and the pivotal structure gives rise to a ribbon category, and thus, a compact closed category, where the resulting twist is given by $\theta_V : |\, n\rangle \mapsto (\mp 1)^{|\, n\,|} \,|\, n\rangle$ for $\tilde{\epsilon}_V : V \otimes V^* \to \mathbb{C} : |\, n\rangle \otimes_{\mathrm{g}} \langle m \,| \mapsto (\pm 1)^{|\, n\,|} \delta_{m,n}$ and corresponding $\tilde{\eta}_V$. Hence, if the right (co)evaluation contains a minus sign, the twist is $\theta_V = \mathrm{id}_V$, which, as mentioned above, is always a valid twist for a symmetric category. However, if the right (co)evaluation contains no minus sign, the twist acts as the parity endomorphism, i.e. as $+1$ on V_0 and as -1 on V_1, which, as we will see in the next section, corresponds to a choice bearing additional structure.

18.5 Adjoints and dagger categories

A final aspect of categories as they are relevant to physics, and in particular quantum physics, is the notion of an adjoint or dagger. A **dagger category** C is a category together with an involutive functor $\dagger : C \to C^{\mathrm{op}}$, i.e. it acts as the identity on objects, whereas on morphisms $f : W \to V$ it defines a morphism $f^\dagger : V \to W$ such that

- $\mathrm{id}_V^\dagger = \mathrm{id}_V$
- $(f \circ g)^\dagger = f^\dagger \circ^{\mathrm{op}} g^\dagger = g^\dagger \circ f^\dagger$
- $(f^\dagger)^\dagger = f$.

Sometimes also the symbol * is used instead of †. However, we have already used * to denote dual objects and transposed morphisms in the case of a pivotal category.

If the category is $\mathbb{C}$-linear, the dagger functor is often assumed to be antilinear, i.e. $(\lambda f)^\dagger = \bar{\lambda} f^\dagger$ for $\lambda \in \mathbb{C}$ and $f \in \mathrm{Hom}(V, W)$. In a dagger category, a morphism $f : W \to V$ is said to be *unitary* if it is an isomorphism and $f^{-1} = f^\dagger$. Furthermore, an endomorphism $f : V \to V$ is *hermitian* or self-adjoint if $f^\dagger = f$. Finally, we will also use the term *isometry* for a morphism $f : W \to V$ which has a left inverse $f^\dagger$, i.e. such that $f^\dagger \circ f = \mathrm{id}_W$, but for which $f \circ f^\dagger$ is not necessarily the identity (but rather some orthogonal projector, i.e. a hermitian idempotent in $\mathrm{End}(V)$).

In the graphical representation, the dagger of a morphism can be represented by mirroring the morphism around a horizontal axis, and then reversing all arrows (bringing them back to their original orientation before the mirror operation):

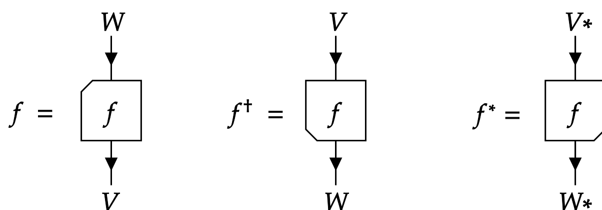


Figure 18.12: dagger

where for completeness we have also depicted the graphical representation of the transpose, which is a very different operation. In particular, the dagger does not reverse the order of the tensor product. Note that, for readibility, we have not mirrored or rotated the label in the box, but this implies that we need to use a type of box for which the action of mirroring or rotating can be observed.

A dagger monoidal category is one in which the associator and left and right unitor are unitary morphisms. Similarly, a dagger braided category also has a unitary braiding, and a dagger balanced category in addition has a unitary twist.

There is more to be said about the interplay between the dagger and duals. Given a left evaluation $\epsilon_V : V^* \otimes V \to I$ and coevaluation $\eta_V : I \to V \otimes V^*$, we can define a right evaluation $\tilde{\epsilon}_V = (\eta_V)^\dagger$ and coevaluation $\tilde{\eta}_V = (\epsilon_V)^\dagger$. Hence, left rigid dagger categories are automatically pivotal dagger categories.

The (right) twist defined via the pivotal structure now becomes

$$\theta_V = (\mathrm{id}_V \otimes (\eta_V)^\dagger) \circ (\tau_{V,V} \otimes \mathrm{id}_{V^*}) \circ (\mathrm{id}_V \otimes \eta_V)$$

and is itself unitary. Even for a symmetric category, the twist defined as such must not be the identity, as we discuss for the $\mathbf{SVect}$ example below.

Finally, the dagger allows to define two Hermitian forms on the morphisms, namely $\langle f, g\rangle_{\mathrm{l/r}} = \mathrm{tr}_{\mathrm{l/r}}(f^\dagger g)$, which coincide for a spherical category. For a *unitary* $\Bbbk$*-linear category*, these Hermitian forms should be positive definite and thus define an inner product on each of the homomorphism spaces $\mathrm{Hom}(W, V)$. In particular then, dimensions of objects are positive, as they satisfy $\dim_{\mathrm{l/r}}(V) = \langle \mathrm{id}_V, \mathrm{id}_V \rangle_{\mathrm{l/r}}$.

This concludes the most important categorical definitions and structures that we want to discuss for the category $\mathbf{Vect}$, but which can also be realized in other categories. In particular, the interface of TensorKit.jl could *in principle* represent morphisms from any $\Bbbk$-linear monoidal category, but assumes categories with duals to be pivotal and in fact spherical, and categories with a braiding to be ribbon categories. A dagger ribbon category where the braiding is symmetric, i.e. a dagger category which is also a compact closed category and where the right (co)evaluation is given via the dagger of the left (co)evaluation, is called a **dagger compact** category. This is the playground of quantum mechanics of bosonic and fermionic systems. However, we also allow for non- symmetric braiding in TensorKit.jl, though this functionality typically requires more careful considerations.

Again studying the category $\mathbf{SVect}_{\mathbb{C}}$ (now explicitly over the complex numbers) and using the conventional adjoint or the complex Euclidean inner product to define the dagger functor, the right (co)evaluation that is obtained from applying the dagger to the left (co)evaluation is the definition we gave above with the $+1$ sign. This choice gives rise to a regular trace (versus the supertrace) of endomorphisms, to positive dimensions, and a non-trivial twist that acts as the parity endomorphism. The resulting category is then a **dagger compact** category, that can be used for the quantum mechanical description of fermionic systems. The bosonic version is obtained by restricting to the subcategory $\mathbf{Vect}$.

18.6 Direct sums, simple objects and fusion categories

These last two section on fusion categories is also applicable, in a straightforward manner, to $\mathbf{Vect}$ and $\mathbf{SVect}$, but is rather meant to provide the background of working with symmetries. We first need two new concepts:

- An object $W \in \mathrm{Ob}(C)$ is a **direct sum** of objects $V_1, V_2, ..., V_k \in \mathrm{Ob}(C)$ if there exists a family morphisms $x_\alpha \in \mathrm{Hom}(V_\alpha, W)$ and $y^\alpha \in \mathrm{Hom}(W, V_\alpha)$ such that $\mathrm{id}_W = \sum_{\alpha=1}^{k} x_\alpha \circ y^\alpha$ and $y^\alpha \circ x_\beta = \delta^\alpha_\beta \mathrm{id}_{V_\alpha}$. The morphisms x_α and y^α are known as *inclusions* and *projections* respectively, and in the context of dagger categories it is natural to assume $y^\alpha = x_\alpha^\dagger$ in order to obtain an orthogonal direct sum decomposition.
- A **simple object** $V \in \mathrm{Ob}(C)$ of a $\Bbbk$-linear category C is an object for which $End_C(V) \approx \Bbbk$, i.e. the algebra of endomorphisms on V is isomorphic to the field (or ring) $\Bbbk$. As $End_C(V)$ always contains the identity morphism id_V, and this must be the only linearly independent endomorphism if V is a simple object, the isomorphism between $\mathrm{End}_C(V)$ and $\Bbbk$ is typically of the form $k \in \Bbbk \leftrightarrow k\mathrm{id}_V \in End_C(V)$. In particular, for $\mathbf{SVect}$ and its subcategory $\mathbf{Vect}$, the unit object I is a simple object.

In particular, for a pivotal $\Bbbk$-linear category where I is simple, it holds that the left and right dimensions of any simple object V are invertible in $\Bbbk$, and that any endomorphism $f \in \mathrm{End}(V)$ can be written as

$$f = (\dim_{\mathrm{l}}(V))^{-1}\mathrm{tr}_{\mathrm{l}}(f)\mathrm{id}_V = (\dim_{\mathrm{r}}(V))^{-1}\mathrm{tr}_{\mathrm{r}}(f)\mathrm{id}_V$$

Strictly speaking, this holds only if the category is non-degenerate, which means that I is simple and that any non-degenerate pairing $e : V \otimes W \to I$ induces a non-degenerate pairing $\mathrm{Hom}(I, V) \otimes \mathrm{Hom}(I, W) \to \mathrm{End}(I)$. This property is always satisfied for a **pre-fusion category** C, i.e. a monoidal $\Bbbk$- linear category having a set $\mathcal{S} \subset \mathrm{Ob}(C)$ of simple objects $\mathcal{S} = \{I, V_1, V_2, ...\}$ such that

- the monoidal unit $I_C \in \mathcal{S}$
- $\mathrm{Hom}_C(V_i, V_j) = 0$ (the singleton set containing only the zero homomorphism) for any distinct $V_i, V_j \in \mathcal{S}$
- every object $V \in \mathrm{Ob}(C)$ can be written as a direct sum of a finite family of elements from $\mathcal{S}$.

Note that in the direct sum decomposition of an object V, a particular simple object V_i might appear multiple times. This number is known as the multiplicity index N_i^V, and equal to the rank of $\mathrm{Hom}(V, V_i)$ or, equivalently, of $\mathrm{Hom}(V_i, V)$. Hence, we can choose inclusion and projection maps $x_{i,\mu} : V_i \to V$ and $y^{i,\mu} : V \to V_i$ for $\mu = 1, ..., N_i^V$, such that $\mathrm{id}_V = \sum_i \sum_{\mu=1}^{N_V^i} x_{i,\mu} \circ y^{i,\mu}$ and $y^{i,\mu} \circ x_{j,\nu} = \delta^i_j \delta^\mu_\nu$. In particular, for a simple object V, it either appears in $\mathcal{S}$ or is isomorphic to an object S. We thus have $N_i^V = 1$ for one particular object V_i and $N_j^V = 0$ for all other j, with x_i and $y^i = (x_i)^{-1}$ representing the isomorphism between V and V_i.

The homomorphisms between two general objects W and V in a pre-fusion category can be decomposed as

$$\mathrm{Hom}(W, V) \approx \bigoplus_{V_i \in \mathcal{S}} \mathrm{Hom}(W, V_i) \otimes \mathrm{Hom}(V_i, V)$$

and thus that the rank of $\mathrm{Hom}(W, V)$ is given by $\sum_i N_i^W N_i^V$.

A **fusion category** is a pre-fusion category that has (left or right) duals, i.e. that is rigid, and that only has a finite number of isomorphism classes of simple objects. Note that the duality functor maps $\mathrm{End}(V)$ to $\mathrm{End}(V^*)$, such that, if V is a simple object, so must be V^*. Henceforth, we will be sloppy about the distinction between a pre-fusion or fusion category, only use the latter term, even when it is not fully justified.

Before continuing, let us use some examples to sketch the relevance of the concept of fusion categories. As mentioned, the categories $\mathbf{Vect}_\Bbbk$ and $\mathbf{SVect}_\Bbbk$ have $I \approx \Bbbk$ as simple object. For $\mathbf{Vect}$, this is the only simple object, i.e. any other vector space V over $\Bbbk$ can be thought of as a direct sum over $N_I^V = \dim(V)$ multiple copies of $\Bbbk$. In $\mathbf{SVect}$, the object $J = 0 \oplus \Bbbk$ with $J_0 = 0$ the zero dimensional space

and $J_1 \approx \mathbb{k}$ is another simple object. Clearly, there are no non-zero grading preserving morphisms between I and J, i. . $\mathrm{Hom}(I, J) = 0$, whereas $\mathrm{Hom}(J, J) \approx \mathbb{k}$. Any other super vector space $V = V_0 \oplus V_1$ can be written as a direct sum over $N_I^V = \dim(V_0)$ copies of I and $N_J^V = \dim(V_1)$ copies of J.

A more representative example is that of the category $C = \mathbf{Rep}_G$, the category of representations of a group G. Colloquially, this could be thought of as a subcategory of $\mathbf{Vect}$ containing as objects vector spaces V on which a representation of G is defined, denoted as $u_V(g)$ for $g \in G$, and as morphisms the equivariant transformations, i.e. intertwiners between the representations on the source and target:

$$\mathrm{Hom}_C(W, V) = \{f \in \mathrm{Hom}_{\mathbf{Vect}}(W, V)|\ u_V(g) \circ f = f \circ u_W(g), \forall g \in G\}.$$

Note that the $u_V(g)$ is itself generally not an element from $End_C(V)$. Simple objects V_a are those corresponding irreducible representations (irreps) a of the group G, for which Schur's lemma implies $End_C(V_a) \approx \mathbb{k}$ and $\mathrm{Hom}_C(V_a, V_b) = 0$ if a and b are not equivalent irreps. On the dual space V^*, the group acts with the contragradient representation, i.e. $u_{V^*}(g) = ((u_V(g))^{-1})^* = u_V(g^{-1})^*$, where one should remind that * denotes the transpose. For a finite group or compact Lie group, we can introduce a dagger and restrict to unitary representations, such that $u_V(g)^{-1} = u_V(g)^\dagger$ and the contragradient representation becomes the complex conjugated representation, denoted as $u_{V^*}(g) = \bar{u}_V(g)$. The resulting category can then be given the structure of a unitary ribbon (pre-)fusion category. (Note that the number of isomorphism classes of simple objects, i.e. the number of non-equivalent irreps, is finite only in the case of a finite group). This example is very relevant to working with symmetries in TensorKit.jl, and will be expanded upon in more detail below.

Fusion categories have a number of simplifying properties. A pivotal fusion category is spherical as soon as $\dim_l(V_i) = \dim_r(V_i)$ (i.e. the trace of the identity morphism) for all (isomorphism classes of) simple objects (note that all isomorphic simple objects have the same dimension). A braided pivotal fusion category is spherical if and only if it is a ribbon category.

18.7 Topological data of a unitary pivotal fusion category

More explicitly, the different structures (monoidal structure, duals and pivotal structure, braiding and twists) in a fusion category can be characterized in terms of the simple objects, which we will henceforth denoted with just a instead of V_a. This gives rise to what is known as the *topological data* of a unitary pivotal fusion category, most importantly the N, F and R symbols, which are introduced in this final section.

Monoidal structure

Starting with the monoidal or tensor product, we start by characterizing how the object $a \otimes b$ can be decomposed as a direct sum over simple objects c, which gives rise to the multiplicity indices N_c^{ab}, as well as the inclusion maps, which we henceforth denote as $X_{c,\mu}^{ab} : c \to a \otimes b$ for $\mu = 1, \ldots, N_{ab}^c$. In the context of a unitary fusion category, on which we now focus, the corresponding projection maps are $Y_{a,b}^{c,\mu} = (X_{c,\mu}^{ab})^\dagger : a \otimes b \to c$ such that

$$(X_{c,\mu}^{ab})^\dagger \circ X_{c',\mu'}^{ab} = \delta_{c,c'}\delta_{\mu,\mu'}\mathrm{id}_c.$$

Graphically, we represent these relations as

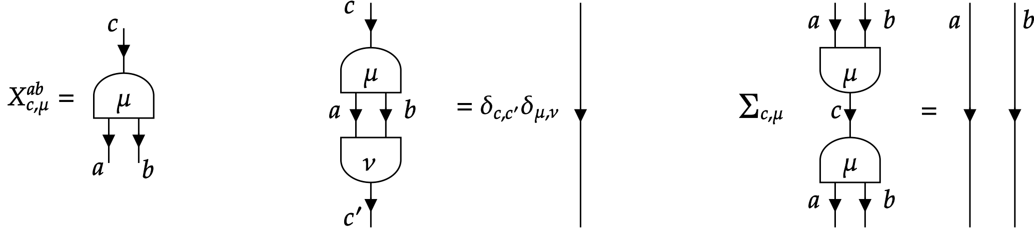


Figure 18.13: fusion

and also refer to the inclusion and projection maps as splitting and fusion tensor, respectively.

For both $(a \otimes b) \otimes c$ and $a \otimes (b \otimes c)$, which are isomorphic via the associator $\alpha_{a,b,c}$, we must thus obtain a direct sum decomposition with the same multiplicity indices, leading to the associativity constraint

$$N_d^{abc} = \sum_e N_e^{ab} N_d^{ec} = \sum_f N_f^{bc} N_d^{af}.$$

The corresponding inclusion maps can be chosen as

$$X^{abc}_{d,(e\mu\nu)} = (X^{ab}_{e,\mu} \otimes \mathrm{id}_c) \circ X^{e,c}_{d\nu} : d \to (a \otimes b) \otimes c.$$

and

$$\tilde{X}^{abc}_{d,(f\kappa\lambda)} = (\mathrm{id}_a \otimes X^{bc}_{f,\kappa}) \circ X^{af}_{d,\lambda} : d \to a \otimes (b \otimes c)$$

and satisfy

$$(X^{abc}_{d,(e\mu\nu)})^\dagger \circ X^{abc}_{d',(e'\mu'\nu')} = \delta_{e,e'}\delta_{\mu,\mu'}\delta_{\nu,\nu'}\delta_{d,d'}\mathrm{id}_d,$$

$$\sum_{d,e\mu\nu} X^{abc}_{d,(e\mu\nu)} \circ (X^{abc}_{d,(e\mu\nu)})^\dagger = \mathrm{id}_{(a\otimes b)\otimes c},$$

and similar for $\tilde{X}^{a,b,c}_{d,(f\kappa\lambda)}$. Applying the associator leads to a relation

$$\alpha_{a,b,c} \circ X^{abc}_{d,(e\mu\nu)} = \sum_{f,\kappa,\lambda} [F^{abc}_d]^{(f\kappa\lambda)}_{(e\mu\nu)} \tilde{X}^{abc}_{d,(f\kappa\lambda)}.$$

which defines the *F-symbol*, i.e. the matrix elements of the associator

$$(\tilde{X}^{abc}_{d,(f\kappa\lambda)})^\dagger \circ \alpha_{a,b,c} \circ X^{abc}_{d',(e\mu\nu)} = \delta_{d,d'}[F^{abc}_d]^{(f\kappa\lambda)}_{(e\mu\nu)}\mathrm{id}_d.$$

Note that the left hand side represents a map in $\mathrm{Hom}(d', d)$, which must be zero if d' is different from d, hence the $\delta_{d,d'}$ on the right hand side. In a strict category, or in the graphical notation, the associator α is omitted and these relations thus represent a unitary basis transform between the basis of inclusion maps $X^{abc}_{d,(e\mu\nu)}$ and $\tilde{X}^{abc}_{d,(f\kappa\lambda)}$, which is also called an F-move, i.e. graphically:

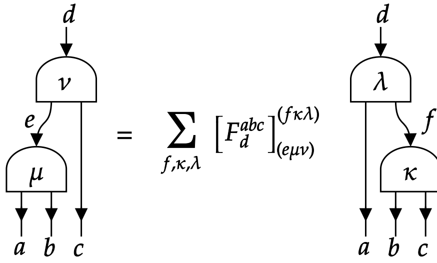


Figure 18.14: Fmove

The matrix F_d^{abc} is thus a unitary matrix. The pentagon coherence equation can also be rewritten in terms of these matrix elements, and as such yields the celebrated pentagon equation for the F-symbols. In a similar fashion, the unitors result in $N_b^{a1} = N_b^{1a} = \delta_b^a$ (where we have now written 1 instead of I for the unit object) and the triangle equation leads to additional relations between the F- symbols involving the unit object. In particular, if we identify $X_{a,1}^{1a} : a \to (1 \otimes a)$ with $\lambda_a^\dagger$ and $X_{a,1}^{a1} : a \to (a \otimes 1)$ with $\rho_a^\dagger$, the triangle equation and its collaries imply that $[F_c^{1ab}]_{(11\mu)}^{(c\nu 1)} = \delta_\mu^\nu$, and similar relations for F_c^{a1b} and F_c^{ab1}, which are graphically represented as

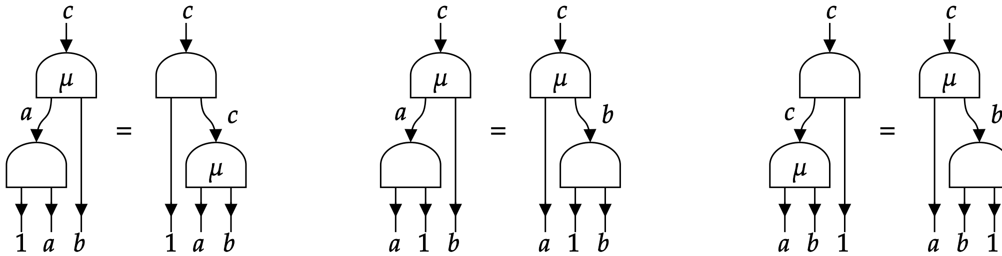


Figure 18.15: Fmove1

In the case of group representations, i.e. the category $\mathbf{Rep}_G$, the splitting and fusion tensors are known as the Clebsch-Gordan coefficients, especially in the case of SU_2. An F-move amounts to a recoupling and the F-symbols can thus be identified with the *6j-symbols* (strictly speaking, Racah's W-symbol for SU_2).

Duality and pivotal structure

Next up is duality. Since we are assuming a dagger category, it can be assumed pivotal, where the left dual objects are identical to the right dual objects, and the left and right (co)evaluation are related via the dagger. We have already pointed out above that the dual object a^* of a simple object a is simple, and thus, it must be isomorphic to one of the representatives $\bar{a}$ of the different isomorphism classes of simple objects that we have chosen. Note that it can happen that $\bar{a} = a$. Duality implies an isomorphism between $\mathrm{Hom}(W, V)$ and $\mathrm{Hom}(I, V \otimes W^*)$, and thus, for a simple object a, $\mathrm{End}(a) \approx \Bbbk$ is isomorphic to $\mathrm{Hom}(1, a \otimes a^*)$, such that the latter is also isomorphic to $\Bbbk$, or thus $N_1^{a\bar{a}} = 1$. Also, all possible duals of a must be isomorphic, and thus there is a single representative $\bar{a}$ such that $N_1^{ab} = \delta^{b,\bar{a}}$, i.e. for all other $b \neq \bar{a}$, $\mathrm{Hom}(1, a \otimes b) \approx \mathrm{Hom}(b^*, a) = 0$. Note that also $\bar{\bar{a}} = a$.

Let us now be somewhat careful with respect to the isomorphism between a^* and $\bar{a}$. If $\bar{a} \neq a$, we can basically choose the representative of that isomorphism class as $\bar{a} = a^*$. However, this choice might not be valid if $\bar{a} = a$, as in that case the choice is already fixed, and might be different from a. To give a concrete example, the $j = 1/2$ representation of SU_2 has a dual (contragradient, but because of unitarity, complex conjugated) representation which is isomorphic to itself, but not equal. In the

context of tensors in quantum physics, we would like to be able to represent this representation and its conjugate, so we need to take the distinction and the isomorphism between them into account. This means that $\mathrm{Hom}(a^*, \bar{a})$ is isomorphic to $\mathbb{k}$ and contains a single linearly independent element, Z_a, which is a unitary isomorphism such that $Z_a^\dagger \circ Z_a = \mathrm{id}_{a^*}$ and $Z_a \circ Z_a^\dagger = \mathrm{id}_{\bar{a}}$. Using the transpose, we obtain $Z_a^* \in \mathrm{Hom}(\bar{a}^*, a)$, and thus it is proportional to $Z_{\bar{a}}$, i.e. $Z_a^* = \chi_a Z_{\bar{a}}$ with χ_a a complex phase (assuming $\mathbb{k} = \mathbb{C}$). Another transpose results in $Z_{\bar{a}}^* = \chi_{\bar{a}} Z_a$ with $\chi_{\bar{a}} = \overline{\chi_a}$, where bar of a scalar quantity denotes its complex conjugate to avoid confusion with the transpose functor. If $a$and $\bar{a}$ are distinct, we can essentially choose $Z_{\bar{a}}$ such that χ_a is 1. However, for $a = \bar{a}$, the value of χ_a cannot be changed, but must satisfy $\chi_a^2 = 1$, or thus $\chi_a = \pm 1$. This value is a topological invariant known as the *Frobenius-Schur indicator*. Graphically, we represent this isomorphism and its relations as

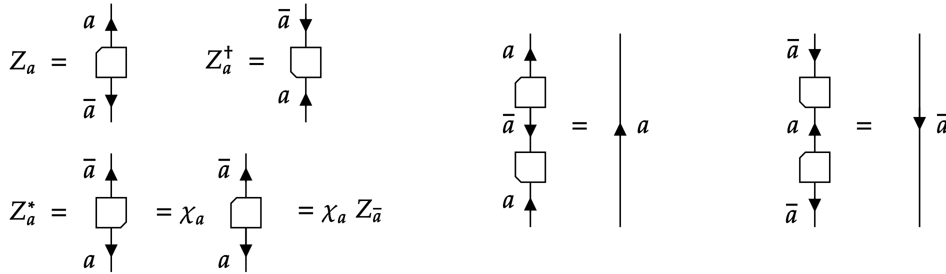


Figure 18.16: Zisomorphism

We can now discuss the relation between the exact pairing and the fusion and splitting tensors. Given that the (left) coevaluation $\eta_a \in \mathrm{Hom}(1, a \otimes a^*)$, we can define the splitting tensor as

$$X_1^{a\bar{a}} = \frac{1}{\sqrt{d_a}}(\mathrm{id}_a \otimes Z_a) \circ \eta_a = \frac{1}{\sqrt{d_a}}(Z_a^* \otimes \mathrm{id}_{\bar{a}}) \circ \tilde{\eta}_{\bar{a}} \in \mathrm{Hom}(1, a \otimes \bar{a}).$$

The prefactor takes care of normalization, i.e. with $\eta_a^\dagger = \tilde{\epsilon}_a$, we find $\eta_a^\dagger \circ \eta_a = \tilde{\epsilon}_a \circ \eta_a = \mathrm{tr}(\mathrm{id}_a) = d_a \mathrm{id}_1$, and thus $(X_1^{a\bar{a}})^\dagger \circ X_1^{a\bar{a}} = \mathrm{id}_1$. Here, we have denoted $d_a = \dim(a) = \mathrm{tr}(\mathrm{id}_a)$ for the quantum dimension of the simple objects a. With this information, we can then compute $[F_a^{a\bar{a}a}]$, which has a single element (it's a 1×1 matrix), and find $[F_a^{a\bar{a}a}] = \frac{\chi_a}{d_a}$, where we've used $\tilde{\eta}_a = \epsilon_a^\dagger$ and the snake rules. Hence, both the quantum dimensions and the Frobenius-Schur indicator are encoded in the F-symbol. Hence, they do not represent new independent data. Again, the graphical representation is more enlightning:

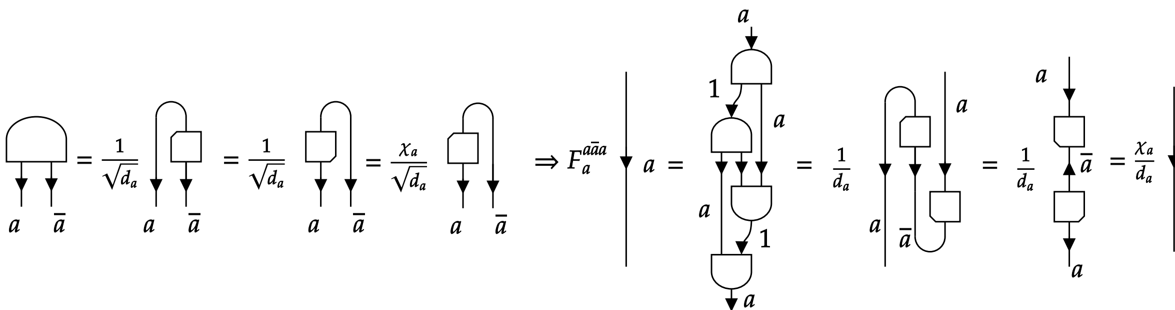


Figure 18.17: ZtoF

With these definitions, we can now also evaluate the action of the evaluation map on the splitting tensors, namely

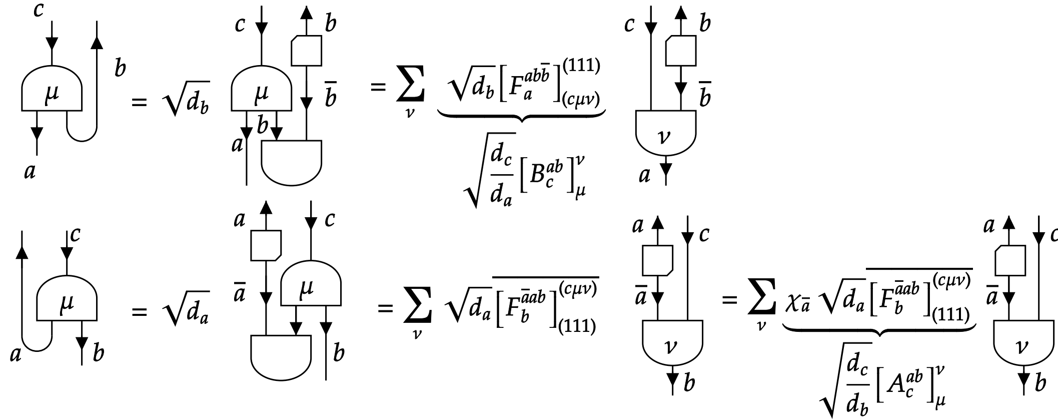


Figure 18.18: splittingfusionrelation

where again bar denotes complex conjugation in the second line, and we introduced two new families of matrices A_c^{ab} and B_c^{ab}, whose entries are composed out of entries of the F-symbol, namely

$$[A_c^{ab}]^\nu_\mu = \sqrt{\frac{d_a d_b}{d_c}} \chi_{\bar{a}} \, \overline{[F_b^{\bar{a}ab}]^{(c\mu\nu)}_{(111)}}$$

and

$$[B_c^{ab}]^\nu_\mu = \sqrt{\frac{d_a d_b}{d_c}} [F_a^{ab\bar{b}}]^{(111)}_{(c\mu\nu)}.$$

Composing the left hand side of first graphical equation with its dagger, and noting that the resulting element $f \in \mathrm{End}(a)$ must satisfy $f = d_a^{-1}\mathrm{tr}(f)\mathrm{id}_a$, i.e.

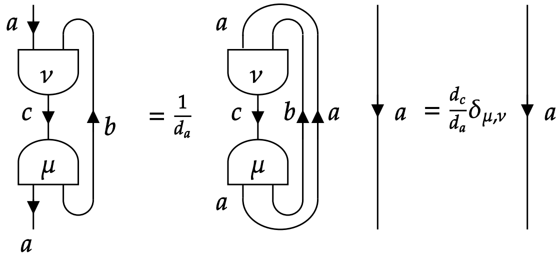


Figure 18.19: Brelation

allows to conclude that $\sum_\nu [B_c^{ab}]^\nu_\mu \overline{[B_c^{ab}]^\nu_{\mu'}} = \delta_{\mu,\mu'}$, i.e. B_c^{ab} is a unitary matrix. The same result follows for A_c^{ab} in analogue fashion.

Note

In the context of fusion categories, one often resorts to the so-called *isotopic* normalization convention, where splitting tensors are normalized as $(X_{c,\mu}^{ab})^\dagger \circ X_{c',\mu'}^{ab} = \sqrt{\frac{d_a d_b}{d_c}} \delta_{c,c'} \delta_{\mu,\mu'} \mathrm{id}_c$. This kills some of the quantum dimensions in formulas like the ones above and essentially allows to rotate the graphical notation of splitting and fusion tensors (up to a unitary transformation). Nonetheless, for our implementation of tensors and manipulations thereof (in particular orthonormal factorizations such as the singular value decomposition), we find it more convenient to work with the original normalization convention.

Let us again study in more detail the example $\mathbf{Rep}_G$. The quantum dimension d_a of an irrep a is just the normal vector space dimension (over $\Bbbk$) of the space on which the irrep acts. The dual of an irrep a is its contragradient representation, which in the case of unitary representations amounts to the complex conjugate representation. This representation can be isomorphic to an already defined irrep $\bar{a}$, for example a itself. If that happens, it does not automatically imply that the irrep a is real-valued. For example, all irreps of SU_2 are self- dual, with the isomorphism given by a π rotation over the y-axis (in the standard basis). The resulting Frobenius-Schur indicator is $+1$ for integer spin irreps, and -1 for half-integer spin irreps. The value $\chi_a = +1$ indicates that the representation can be made real, e.g. the integer spin representations can be written as tensor representations of SO_3 by a change of basis. The value $\chi_a = -1$ indicates that the representation is quaternionic and cannot be made real.

The (co)evaluation expresses that the standard contraction of a vector with a dual vector yields a scalar, i.e. a representation and its dual (the contragradient) yields the trivial representation when correctly contracted. The coevaluation together with the isomorphism between the conjugate of irrep a and some irrep $\bar{a}$ yields a way to define the Clebsch-Gordan coefficients (i.e. the splitting and fusion tensor) for fusing $a \otimes \bar{a}$ to the trivial irrep, i.e. to what is called a singlet in the case of SU_2.

Braidings and twists

Finally, we can study the braiding structure of a pivotal fusion category. Not all fusion categories have a braiding structure. The existence of a braiding isomorphism $\tau_{V,W} : V \otimes W \to W \otimes V$ requires at the very least that $N_c^{ab} = N_c^{ba}$ at the level of the simple objects. We can then express $\tau_{a,b}$ in terms of its matrix elements as

$$\tau_{a,b} \circ X_{c,\mu}^{ab} = \sum_\nu [R_c^{ab}]_\mu^\nu X_{c,\nu}^{ba}$$

or graphically

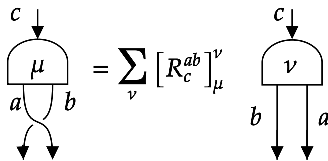


Figure 18.20: braidingR

The hexagon coherence axiom for the braiding and the associator can then be reexpressed in terms of the F-symbols and R-symbols.

We can now compute the twist, which for simple objects needs to be scalars (or in fact complex phases because of unitarity) multiplying the identity morphism, i.e.

$$\theta_a = \mathrm{id}_a \sum_{b,\mu} \frac{d_b}{d_a} [R_b^{aa}]_\mu^\mu$$

or graphically

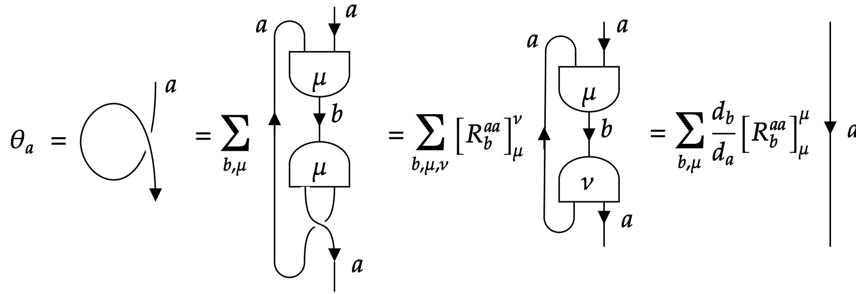


Figure 18.21: simpletwist

Henceforth, we reserve θ_a for the scalar value itself. Note that $\theta_a = \theta_{\bar{a}}$ as our category is spherical and thus a ribbon category, and that the defining relation of a twist implies

$$[R^{ba}_c]^{\kappa}_{\mu}[R^{ab}_c]^{\mu}_{\nu} = \frac{\theta_c}{\theta_a\theta_b}\delta^{\kappa}_{\nu}$$

If $a = \bar{a}$, we can furthermore relate the twist, the braiding and the Frobenius- Schur indicator via $\theta_a \chi_a R^{aa}_1 = 1$, because of

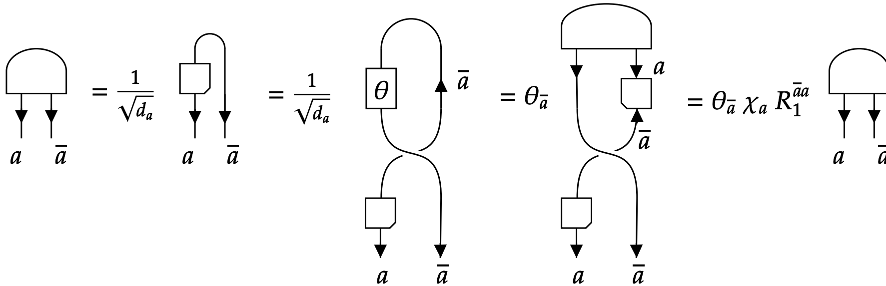


Figure 18.22: twistfrobeniusschur

For the recurring example of $\mathbf{Rep}_G$, the braiding acts simply as the swap of the two vector spaces on which the representations are acting and is thus symmetric, i.e. $\tau_{b,a} \circ \tau_{a,b} = \mathrm{id}_{a\otimes b}$. All the twists are simply $\theta_a = 1$. For an irrep that is self-dual, i.e. $\bar{a} = a$, the final expression simplifies to $R^{aa}_1 = \chi_a$ and thus states that the fusion from $a \otimes a$ to the trivial sector is either symmetric under swaps if $\chi_a = 1$ or antisymmetric if $\chi_a = -1$. For the case of SU_2, the coupling of two spin j states to a singlet it symmetric for integer j and odd for half-integer j.

With this, we conclude our exposition of unitary fusion categories. There are many fusion categories that do not originate from the representation theory of groups, but are related to quantum groups and the representation theory of quasi-triangular Hopf algebras. They have non-integer quantum dimensions and generically admit for braidings which are not symmetric. A particular class of interesting fusion categories are *modular fusion categories*, which provide the mathematical structure for the theory of anyons and topological sectors in topological quantum states of matter. Thereto, one defines the modular S matrix, defined as

$$S_{a,b} = \frac{1}{D}\mathrm{tr}(\tau_{a,b} \circ \tau_{b,a}) = \frac{1}{D}\sum_c N^{ab}_c d_c \frac{\theta_c}{\theta_a\theta_b}.$$

The normalization constant is given by $D = \sqrt{\sum_a d_a^2}$, and thus truly requires a fusion category with a finite number of (isomorphism classes of) simple objects. For a modular fusion category, the symmetric matrix S is non-degenerate, and in fact (for a unitary fusion category) unitary. Note, however, that for

a symmetric braiding $S_{a,b} = \frac{d_a d_b}{D}$ and thus S is a rank 1 matrix. In particular, $\mathbf{Rep}_G$ is never a modular category and the properties associated with this are not of (direct) importance for TensorKit.jl. We refer to the references for further information about modular categories.

18.8 Bibliography

Part VI

Changelog

Chapter 19

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

19.1 Guidelines for updating this changelog

When making changes to this project, please update the "Unreleased" section with your changes under the appropriate category:

- **Added** for new features.
- **Changed** for changes in existing functionality.
- **Deprecated** for soon-to-be removed features.
- **Removed** for now removed features.
- **Fixed** for any bug fixes.
- **Performance** for performance improvements.

When releasing a new version, move the "Unreleased" changes to a new version section with the release date.

19.2 Unreleased

Added

Changed

Deprecated

Removed

Fixed

19.3 0.16.3 - 2026-02-22

Added

- Expanded set of Mooncake AD rules (#356)
- Adapt support for `BraidingTensor` (#374)

Changed

- Documentation improvements and updates (#345)

Fixed

- Small fixes for upstream compatibility and CUDA support, including `Base.ones` / `zeros` accepting `CuArray` (#373)

19.4 0.16.2 - 2026-02-10

Added

- A more robust promotion system for `storagetype` s to better handle working with unions and other abstract tensor map types (#370).

Fixed

- Fix `findtruncated` with `truncspace` (#369)
- Fix `truncrank` when kept rank is larger than input (#368)
- Added missing `similar` definition for `SectorVector` (#367)
- Small fixes for CUDA support (#366)

19.5 0.16.1 - 2026-02-05

Added

- Extended support for selecting storage types in the `TensorMap` constructors (#327)
- `similar_diagonal` to handle storage types when constructing diagonals (#330)
- Support for CUDA.jl (#336,#325)
- Support for Adapt.jl (#344)
- Preliminary support for Mooncake (#352)
- Export `TimeReversed` symbol (#337)

Fixed

- Issue with using relative tolerances in truncation schemes (#314)
- Using `scalartype` instead of `eltype` in BLAS contraction (#326)
- Divide by zero error in `show` for empty tensors (#329)
- `svd_vals(::DiagonalTensorMap)` correctly outputs `SectorVector` and implementation fix. (#333)
- Fix handling of real tensors with complex scalartype (#360)
- Sorted diagonal eigenvalues to ensure consistent ordering (#350)
- Adding tensors of different types now correctly promotes (#364)

Changed

- `convert(TensorMap, t)` now retains `storagetype` when converting (#357)
- `transpose` specialization for `DiagonalTensorMap` for improved correctness/performance (#335)

- Uniformized `CartesianSpace` and `ComplexSpace` constructors (#334)

Performance

- GPU-friendly truncation implementations (#349)
- `norm` performance optimizations (#351)
- TensorOperations ChainRules performance improvements (#343)
- Type-stability and small test fixes (various commits)

19.6 0.16.0 - 2025-12-08

Added

- `rrule` for `transpose` operation (#319)
- New functions for multifusion support: `unitspace` , `zerospace` , `leftunitspace` , `rightunitspace` , `isunitspace` (#291)
- Support for projections and orthogonal complements (#312)

Changed

- Improvements to the default printing of tensors, where only a (possibly compressed) representation of the (possibly truncated) list of diagonal blocks is printed. Use `blocks(t)` and `subblocks(t)` for a full inspection of the tensor data (#304, #322))
- Updated `left_orth` , `right_orth` , `left_null` and `right_null` interfaces for MatrixAlgebraKit v0.6 (#312)
- Updated `ishermitian` and `isisometric` implementations (#312)
- Sector functions now by default use `unit` instead of `one` , `isunit` instead of `isone` , and `dual` instead of `conj` (#291)
- Reworked TensorOperations implementation to use backend and allocator system (#311)
- Major documentation update/overhaul (#289)
- Added symmetric tensor tutorial as appendix (#316)
- Improved error messages throughout codebase (#309)
- `eigvals` and `svdvals` now output `SectorVector` objects, which do behave as `AbstractVector` but also have the option of iterating the blocks through `Base.pairs` . (#324

Deprecated

Removed

- All deprecations from v0.15: old factorization function names (`leftorth` , `rightorth` , `tsvd` , `eig` , `eigh`)
- Old truncation strategy names (`truncdim` , `truncbelow`)
- Old factorization struct types (`OrthogonalFactorization`)
- Old constructor syntaxes and deprecated `rand*` function names

Fixed

- Avoid unnecessary copy in `twist` for tensors with bosonic braiding (#305)
- Small fixes and typos (#295)
- `eig_vals` , `svd_vals` , etc now all output `SectorVector` objects instead of `DiagonalTensorMap` s, in line with how MatrixAlgebraKit returns `Vector` s instead of `Diagonal` s (#324

19.7 0.15.3 - 2025-10-30

Fixed

- Fixed typo in `show(::GradedSpace)` (#308)
- Updated printing of `ProductSpace{<:Any,0}`
- Added tests for show methods

19.8 0.15.2 - 2025-10-28

Added

- `subblocks` iterator for easier inspection of tensor data (#304)

Changed

- Tensors no longer print their data by default, only their spaces. Use `blocks(t)` or `subblocks(t)` to inspect data (#304)
- Updated compatibility to TensorKitSectors v0.3 (#290)
- Refactored test suite and split into groups (#298)

Fixed

- Fixed `TruncationIntersection` implementation and test (#300)
- Avoided unnecessary allocations in rrules for contractions and tensor products (#306)

19.9 0.15.1 - 2025-10-09

Fixed

- Small fixes and typo corrections (#295)

19.10 0.15.0 - 2025-10-03

Added

- MatrixAlgebraKit as new backend for tensor factorizations (#230)
- `foreachblock(f, t::AbstractTensorMap...)` - uniform interface to iterate through tensor blocks
- `eig_trunc` and `eigh_trunc` - truncated eigenvalue decompositions

- `ominus` (and unicode `⊖`) - compute orthogonal complement of a space
- Backend selection for factorizations - swap algorithms or implementations

Changed

- `left_orth` and `right_orth` now always output tensors with a single connecting space
- `left_orth` and `right_orth` now always have connecting space with `isdual=false`
- Code formatter is now Runic.jl

Deprecated

- Factorization functions `leftorth` , `rightorth` , `tsvd` , `eig` , `eigh` in favor of MatrixAlgebraKit variants (`left_orth` , `right_orth` , `svd_compact` , `eig_full` , `eigh_full`)
- Truncation strategies: `truncdim` (use `truncrank`) and `truncbelow` (use `trunctol`)
- `OrthogonalFactorization` structs (constructors deprecated to return equivalent MatrixAlgebraKit algorithm structs)

Removed

- Direct permute-and-factorize operations (incompatible with `permute` vs `braid` distinction)
- `Polar` decomposition behavior for `left_orth` / `right_orth` (use `left_polar` / `right_polar` instead for `isposdef` R factors)

19.11 0.14.0 - 2024-12-19

Added

- `DiagonalTensorMap` type for representing tensor maps with diagonal blocks
- `reduceddim(V)` function that sums up degeneracy dimensions for each sector
- New index manipulation functions:
  - `flip(t, i)`
  - `insertleftunit(t, i)`
  - `insertrightunit(t, i)`
  - `removeunit(t, i)`

Changed

- Singular values and eigenvalues now explicitly represented as `DiagonalTensorMap` instances
- SVD truncation now guarantees smaller singular values are removed first, irrespective of sector quantum dimension

19.12 0.13.0 - 2024-11-24

Added

- Refactored `TensorMap` constructors to align with Julia `Array` constructors
- Convenience constructors: `ones` , `zeros` , `rand` , `randn` for tensors

- TensorOperations v5 support

Changed

- Scalar type as parameter to `AbstractTensorMap` type: `AbstractTensorMap{E, S, N₁, N₂}`
- Default way to create uninitialized tensors is now `TensorMap{E}(undef, codomain ← domain)`
- Behavior of `copy` for `BraidingTensor` to properly instantiate a `TensorMap`
- TensorKitSectors promoted to separate package
- `TensorMap` data structure now consists of single vector with blocks as views
- `FusionTree` vertices now only use `Int` labels for `GenericFusion` sectors