Functions

The elementary tensor operations can also be accessed via functions, mainly for compatibility with older versions of this toolbox. The function-based syntax is also required when the contraction pattern is not known at compile time but is rather determined dynamically.

These functions come in a mutating and non-mutating version. The mutating versions mimick the argument order of some of the BLAS functions, such as `blascopy!`, `axpy!` and `gemm!`. Symbols `A` and `B` always refer to input arrays, whereas `C` is used to denote the array where the result will be stored. They also return `C` and are therefore type stable. The greek letters `α` and `β` denote scalar coefficients.

`TensorOperations.tensorcopy!` — Function

```
tensorcopy!(A, IA, C, IC)
```

Copies `A` into `C` by permuting the dimensions according to the pattern specified by `IA` and `IC`. Both iterables should contain the same elements in a different order. The result of this method is equivalent to `permutedims!(C, A, p)` where `p` is the permutation such that `IC=IA[p]`. The implementation of tensorcopy! is however more efficient on average, especially if `Threads.nthreads() > 1`.

`TensorOperations.tensoradd!` — Function

```
tensoradd!(α, A, IA, β, C, IC)
```

Updates `C` to `β*C + α * tensorcopy(A,IA,IC)`, but without creating the temporary permuted array.

See also: `tensorcopy`

`TensorOperations.tensortrace!` — Function

```
tensortrace!(α, A, IA, β, C, IC)
```

Updates `C` to `β*C + α tensortrace(A,IA,IC)`, but without creating the temporary traced array.

See also: `tensortrace`

`TensorOperations.tensorcontract!` — Function

```
tensorcontract!(α, A, labelsA, conjA, B, labelsB, conjB, β, C, labelsC)
```

Replaces `C` with `β C + α A * B`, where some indices of array `A` are contracted with corresponding indices in array `B` by assigning them identical labels in the iterables `labelsA` and `labelsB`. The arguments `conjA` and `conjB` should be of type `Char` and indicate whether the data of arrays `A` and `B`, respectively, need to be conjugated (value `'C'`) or not (value `'N'`). Every label should appear exactly twice in the union of `labelsA`, `labelsB` and `labelsC`, either in the intersection of `labelsA` and `labelsB` (for indices that need to be contracted) or in the interaction of either `labelsA` or `labelsB` with `labelsC`, for indicating the order in which the open indices should be match to the indices of the output array `C`.

`TensorOperations.tensorproduct!` — Function

```
tensorproduct!(α, A, labelsA, B, labelsB, β, C, labelsC)
```

Replaces C with `β C + α A * B` without any indices being contracted.

The non-mutating functions are simpler in not allowing scalar coefficients and conjugation. They also take a default value for the labels of the output array if these are not specified. However, the return type is only inferred if the labels are entered as tuples, and also `IC` is specified. They are simply called as:

`TensorOperations.tensorcopy` — Function

```
tensorcopy(A, IA, IC = IA)
```

Creates a copy of `A`, where the dimensions of `A` are assigned indices from the iterable `IA` and the indices of the copy are contained in `IC`. Both iterables should contain the same elements in a different order.

The result of this method is equivalent to `permutedims(A, p)` where p is the permutation such that `IC = IA[p]`. The implementation of `tensorcopy` is however more efficient on average, especially if

Threads.nthreads() > 1.

`TensorOperations.tensoradd` — Function

```
tensoradd(A, IA, B, IB, IC = IA)
```

Returns the result of adding arrays `A` and `B` where the iterabels `IA` and `IB` denote how the array data should be permuted in order to be added. More specifically, the result of this method is equivalent to

```
tensorcopy(A, IA, IC) + tensorcopy(B, IB, IC)
```

but without creating the temporary permuted arrays.

`TensorOperations.tensortrace` — Function

```
tensortrace(A, IA [, IC])
```

Trace or contract pairs of indices of array `A`, by assigning them an identical indices in the iterable `IA`. The untraced indices, which are assigned a unique index, can be reordered according to the optional argument `IC`. The default value corresponds to the order in which they appear. Note that only pairs of indices can be contracted, so that every index in `IA` can appear only once (for an untraced index) or twice (for an index in a contracted pair).

`TensorOperations.tensorcontract` — Function

```
tensorcontract(A, IA, B, IB[, IC])
```

Contract indices of array `A` with corresponding indices in array `B` by assigning them identical labels in the iterables `IA` and `IB`. The indices of the resulting array correspond to the indices that only appear in either `IA` or `IB` and can be ordered by specifying the optional argument `IC`. The default is to have all open indices of array `A` followed by all open indices of array `B`. Note that inner contractions of an array should be handled first with `tensortrace`, so that every label can appear only once in `IA` or `IB` seperately, and once (for open index) or twice (for contracted index) in the union of `IA` and `IB`.

The contraction can be performed by a native Julia algorithm without creating any temporaries, or by first permuting the arrays such that the contraction becomes equivalent to a matrix product, which is then performed by BLAS. The latter is typically faster for large arrays. The choice of method is globally controlled by the methods `enable_blas()` and `disable_blas()`.

`TensorOperations.tensorproduct` — Function

```
tensorproduct(A, IA, B, IB, IC = (IA..., IB...))
```

Computes the tensor product of two arrays `A` and `B`, i.e. returns a new array `C` with `ndims(C) = ndims(A)+ndims(B)`. The indices of the output tensor are related to those of the input tensors by the pattern specified by the indices. Essentially, this is a special case of `tensorcontract` with no indices being contracted over. This method checks whether the indices indeed specify a tensor product instead of a genuine contraction.

« Index notation with macros Cache for temporaries »