abeliantensors

Apr 27, 2021

Contents:

1 Symmetric tensor classes 3

2 AbelianTensor 5

3 Tensor 11

4 TensorCommon 15

Index 19

abeliantensors is a Python 3 package for symmetric tensors, as used in tensor network algorithms. For installation instructions, usage examples, and other information, see the README at github.com/mhauru/abeliantensors. This page contains the API reference of the package.

- genindex

CHAPTER 1

Symmetric tensor classes

class abeliantensors.symmetrytensors.**TensorU1**(*shape*, *qhape=None*, *qodulus=None*, *sects=None*, *dirs=None*, *dtype=<class 'numpy.float64'>*, *defval=0*, *charge=0*, *invar=True*)

Bases: *abeliantensors.abeliantensor.AbelianTensor*

A class for U(1) symmetric tensors.

See the parent class *AbelianTensor* for details.

class abeliantensors.symmetrytensors.**TensorZ2**(*shape*, **args*, *qhape=None*, *qodulus=None*, ***kwargs*)

Bases: *abeliantensors.symmetrytensors.TensorZN*

A class for Z2 symmetric tensors.

See the parent class *AbelianTensor* for details.

class abeliantensors.symmetrytensors.**TensorZ3**(*shape*, **args*, *qhape=None*, *qodulus=None*, ***kwargs*)

Bases: *abeliantensors.symmetrytensors.TensorZN*

A class for Z3 symmetric tensors.

See the parent class *AbelianTensor* for details.

class abeliantensors.symmetrytensors.**TensorZN**(*shape*, **args*, *qhape=None*, *qodulus=None*, ***kwargs*)

Bases: *abeliantensors.abeliantensor.AbelianTensor*

A symmetric tensor class for the cyclic group of order N.

See *AbelianTensor* for the details: A *TensorZN* is just an *AbelianTensor* for which addition of charges is done modulo N.

classmethod eye(*dim*, *qim=None*, *qodulus=None*, *dtype=<class 'numpy.float64'>*)

Return the identity matrix of the given dimension *dim*.

classmethod from_ndarray(*a*, **args*, *shape=None*, *qhape=None*, *qodulus=None*, ***kwargs*)

Build a *TensorZN* out of a given NumPy array, using the provided form data.

If *qhape* is not provided, it is automatically generated based on *shape* to be `[0, ..., N]` for each index. See *AbelianTensor.from_ndarray* for more documentation.

classmethod initialize_with(*numpy_func*, *shape*, **args*, *qhape=None*, *qodulus=None*, ***kwargs*)

Return a tensor of the given *shape*, initialized with *numpy_func*.

split_indices(*indices*, *dims*, *qims=None*, *dirs=None*)

Split indices in the spirit of reshape.

If *qhape* is not provided, it is automatically generated based on *shape* to be `[0, ..., N]` for each index. See *AbelianTensor.split* for more documentation.

CHAPTER 2

AbelianTensor

class abeliantensors.abeliantensor.**AbelianTensor**(*shape*, *qhape=None*, *qodulus=None*, *sects=None*, *dirs=None*, *dtype=<class 'numpy.float64'>*, *defval=0*, *charge=0*, *invar=True*)

Bases: *abeliantensors.tensorcommon.TensorCommon*

A class for symmetry preserving tensors capabable of handling abelian symmetry groups.

This class is meant to be subclassed to implement specific symmetries, which can typically be done by simply fixing the *qodulus* of the class (see below).

Every *AbelianTensor* has the following attributes:

shape: A list of *dims*, one *dim* per index. Every *dim* is a list of integerers that are the dimensions of the different quantum number blocks along that indices.

qhape: A list of *qims*, one *qim* per index. Every *qim* is a list of unique integers that are the quantum numbers (*qnums*), aka charges, of that index. The quantum numbers are in one-to-one correspondence with the elements of the *dims*, so that `qhape[i][j]` and `shape[i][j]` are the *qnum* and dimension of the same block.

dirs: A list of integers -1 or 1, one for each index. 1 means that the corresponding index is outgoing, -1 means incoming.

qodulus: An integer or None. If an integer, then all arithmetic on the quantum numbers is done modulo *qodulus*. If None then arithmetic on *qnums* is just usual integer arithmetic.

sects: A dict of numpy arrays, with combinations of quantum numbers as keys. Every key must a tuple of quantum numbers, one for each index, and each one of them being from the *qim* of that index. The value of the dict at this key is the block (or "sector" or "sect") corresponding to these quantum numbers. If the tensor is invariant under a symmetry (see *invar*) then only certain blocks are allowed to be set, but even in such a cause not all allowed blocks must be set. For the treatement of unset blocks see *defval*.

dtype: A NumPy dtype, that is the dtype of all the sects.

defval: The default value that the tensor has everywhere outside the blocks set in *sects*. If the tensor is a scalar with no indices then its value is its *defval* and it has no blocks. Note that many of the methods - such as *dot* and

svd - require `defval == 0` (and assert this). The main use of `defval != 0` is to be able to handle tensors of boolean values that arise in comparisons.

charge: An integer such that if *invar* is True then all the blocks set in *sects* must have keys *k* such that `sum_i k[i]*dirs[i] % qodulus == charge`.

invar: A boolean. If True, then the tensor is invariant under the symmetry determined by *qodulus*, in the sense described in the definition of *charge*. If False, this condition is ignored and any block can be set. Note that as with *defval*, many methods require the tensor to be invariant and `invar == False` is mainly used for handling vectors of singular values and eigenvalues. If `invar == True` then *defval* must be 0, unless the tensor is a scalar of *charge* 0.

Note that many of these rules are not constantly checked for and can be broken by the user. In such cases behavior of the class is not guaranteed. The method *check_consistency* can be used to check that the tensor conforms to this definition.

all()
Check whether all of the elements of the tensor are True.

allclose(*B*, *rtol=1e-05*, *atol=1e-08*)
Check whether all of the elements of the two tensors are close to each other.

See *numpy.allclose* for explanations of the tolerance arguments.

any()
Check whether any of the elements of the tensor are True.

astype(*dtype*, *casting='unsafe'*, *copy=True*)
Change the dtype of the tensor.

By default creates a copy, but works in place if `copy=False`.

average()
Return the average of all elements.

check_consistency()
Check internal consistency of a tensor.

Check that self conforms to the defition given in the documentation of the class. If yes, return True, otherwise raise an *AssertionError*. This method is meant to be used by the user (probably for debugging) and is not called anywhere in the class.

classmethod check_form_match(*tensor1=None*, *tensor2=None*, *qhape1=None*, *shape1=None*, *dirs1=None*, *qhape2=None*, *shape2=None*, *dirs2=None*, *qodulus=None*)
Check that the form data of two tensors match.

Check that the given two tensors have the same form in the sense that if their indices are all flipped to point in the same direction then both tensors have the same *qnums* for the same indices and with the same dimensions. Instead of giving two tensors, sets of *qhapes*, *shapes*, and *dirs* and a *qodulus* can also be given.

classmethod check_qhape_shape_match(*qhape*, *shape*)
Check that the given *qhape* and *shape* match, i.e. are valid for the same tensor.

classmethod check_qim_dim_match(*qim*, *dim*)
Check that the given *qim* and *dim* match, i.e. are valid for the same index.

compatible_indices(*other*, *i*, *j*)
Return True if index *i* of *self* may be contracted with index *j* of *other*, False otherwise.

Flipping of indices is allowed (but not done, this is only a check).

conj()
Return a new tensor that is the complex conjugate of this one, with the directions of all the indices flipped and the charge of negated.

conjugate()
Return a new tensor that is the complex conjugate of this one, with the directions of all the indices flipped and the charge of negated.

copy(*memo=None*, *_nil=[]*)
Deep copy operation on arbitrary Python objects.

See the module's __doc__ string for more info.

defblock(*key*)
Return an NumPy array of the size of the block `self[key]`, filled with *self.defval*.

This works regardless of whether `self[key]` is set or not and whether the block is allowed by symmetry.

diag()
Either map a square matrix to a vector of its diagonals or a vector to diagonal square matrix.

If the input is a vector (which may be non-invariant) with `qhape = [qim]`, `shape = [dim]` and `dir = [d]`, then the output is an invariant matrix with `qhape = [qim, qim]`, `shape = [dim, dim]` and `dirs = [d, -d]`.

If the input is a matrix it should be invariant and square in the sense that its two indices are compatible, i.e. could be contracted with each other. If `self.dirs == [d, d]` then the latter is flipped and a warning is raised. The output is then a non-invariant vector with `dirs = [d]`.

empty_like()
Initialize a tensor that is like a copy of this one, but with an empty sects.

expand_dims(*axis*, *direction=1*)
Return a view of *self* that has an additional index at the position *axis*.

This new index has only one *qnum*, 0, and dimension 1. The direction of the new index is a keyword argument *direction* that defaults to 1.

classmethod eye(*dim*, *qim=None*, *qodulus=None*, *dtype=<class 'numpy.float64'>*)
Return an identity tensor of `shape = [dim, dim]`, `qhape = [qim, qim]` and `dirs = [1, -1]`.

fill(*value*)
Set all the elements of the tensor to be *value*.

This really means all, not just the ones in allowed blocks.

flip_dir(*axis*)
Flip the direction of the given *axis* of *self*.

The operation is not in-place, but a view is returned. The quantum numbers along given axis are also negated accordingly, so that the tensor as a whole remains the same.

classmethod from_ndarray(*a*, *shape=None*, *qhape=None*, *dirs=None*, *qodulus=None*, *invar=True*, *charge=0*)
Build an *AbelianTensor* out of a given NumPy array, using the provided form data.

Although *shape* and *qhape* are keyword arguments to maintain a common interface with *Tensor*, they are not optional. The blocks are read in the same order as they are written in *to_ndarray*, i.e. rising *qnum* along every index. Note hence that the ordering of the *qnums* in the *qhape* given has no effect.

imag()
Return the imaginary part.

classmethod initialize_with(*numpy_func*, *shape*, **args*, *qhape=None*, *qodulus=None*, *dirs=None*, *invar=True*, *charge=0*, ***kwargs*)
Create a tensor initialized with a given numpy function.

initialize_with will be called with different *numpy_funcs* to create initializer functions such as *zeros* and *random*. It sets all the valid blocks of the new tensor to `numpy_func(block_shape, *args, **kwargs)`.

is_full()
Return True if the elements in *self.sects* cover all the elements in *self*.

is_valid_key(*key*)
Return True if *key* is a valid block allowed by symmetry or *self.invar* is False. Otherwise False.

isscalar()
Return True is this tensor is scalar, False otherwise.

join_indices(**inds*, *dirs=None*, *return_transposed_shape_data=False*)
Join indices together in the spirit of reshape.

inds is either an iterable of indices, in which case they are joined, or an iterable of iterables of indices, in which case the indices listed in each element of *inds* (a "batch") will be joined. So for instance `inds=[[0,1], [2,3]]` causes the joining of both 0 and 1, and of 2 and 3, at the same time.

Before any joining is done the indices are transposed so that for every batch of indices to be joined the first remains in place and the others are moved to be after it in the order given. The order in which the batches are given does not matter.

dirs are the directions of the new indices, defaults to `[1,...,1]`. If a batch of indices to be joined consists of only one index, its direction will be flipped to be as in *dirs*.

If *return_transposed_shape_data* is True, then the *shape*, *qhape* and *dirs* (in this order) of the tensor after transposing but before reshaping are returned as well.

The method does not modify the original tensor, but returns a copy or a view.

matrix_dot(*other*)
Take the dot product of two tensors of order < 3.

If either one is a matrix, it must be invariant and have `defval == 0`.

matrix_eig(*chis=None*, *eps=0*, *print_errors='deprecated'*, *hermitian=False*, *break_degenerate=False*, *degeneracy_eps=1e-06*, *sparse=False*, *trunc_err_func=None*)
Find eigenvalues and eigenvectors of a matrix.

The input must have `defval == 0`, `invar == True`, `charge == 0`, and must be square in the sense that the dimensions must have the same *qim* and *dim* and opposing *dirs*.

If *hermitian* is True the matrix is assumed to be hermitian.

Truncation works like for SVD, see the docstring there for more.

If *sparse* is True, a sparse eigenvalue decomposition, using power methods from *scipy.sparse.eigs* or *eigsh*, is used. This decomposition is done to find `max(chis)` eigenvalues, after which the decomposition may be truncated further if the truncation error so allows. Thus `max(chis)` should be much smaller than the full size of the matrix, if *sparse* is True.

The return value is `S, U, rel_err`, where *S* is a non-invariant vector of eigenvalues and *U* is a matrix that has as its columns the eigenvectors. Both have the same *dim* and *qim* as self. *rel_err* is the truncation error.

matrix_svd(*chis=None*, *eps=0*, *print_errors='deprecated'*, *break_degenerate=False*, *degeneracy_eps=1e-06*, *sparse=False*, *trunc_err_func=None*)
Singular value decompose a matrix.

The matrix must have `invar == True` and `defval == 0`.

The optional argument *chis* is a list of bond dimensions. The SVD is truncated to one of these dimensions *chi*, meaning that only *chi* largest singular values are kept. If *chis* is a single integer (either within a singleton list or just as a bare integer) this dimension is used. If `eps == 0`, the largest value in *chis* is used. Otherwise the smallest *chi* in *chis* is used, such that the relative error made in the truncation is smaller than *eps*. The truncation error is by default the Frobenius norm of the difference, but can be specified with the keyword agument *trunc_err_func*.

An exception to the above is made by degenerate singular values. By default truncation is never done so that some singular values are included while others of the same value are left out. If this is about to happen, *chi* is decreased so that none of the degenerate singular values are included. This default behavior can be changed with the keyword argument *break_degenerate*. The default threshold for when singular values are considered degenerate is 1e-6. This can be changed with the keyword argument *degeneracy_eps*.

If *sparse* is True, a sparse SVD, using power methods from *scipy.sparse.svds*, is used. This SVD is done to find `max(chis)` singular values, after which the decomposition may be truncated further if the truncation error so allows. Thus `max(chis)` should be much smaller than the full size of the matrix, if *sparse* is True.

The method returns the tuple `U, S, V, rel_err`, where *S* is a non-invariant vector and *U* and *V* are unitary matrices. They are such that `U.diag(S).V == self`, where the equality is appromixate if there is truncation. *U* and *S* have always charge 0, but *V* has the same charge as *self*. *U* has *dirs* `[d, -d]` where `d = self.dirs[0]`, but *V* has the same *dirs* as *self*. *rel_err* is the truncation error.

max()
Return the maximum element.

min()
Return the minimum element.

multiply_diag(*diag_vect*, *axis*, *direction='r'*)
Multiply by a diagonal matrix on one axis.

The result of *multiply_diag* is the same as `self.dot(diag_vect.diag(), (axis, 0))` if *direction* is "right" or "r" (the diagonal matrix comes from the right) or `self.dot(diag_vect.diag(), (axis, 1))` if *direction* is "left" or "l". This operation is just done without constructing the full diagonal matrix.

real()
Return the real part.

split_indices(*indices*, *dims*, *qims=None*, *dirs=None*)
Split indices in the spirit of reshape.

indices is an iterable of indices to be split. *dims* is an iterable such that `dim_batch=dims[i]` is an iterable of lists of dimensions, each list giving the dimensions along a new index that will come out of splitting `indices[i]`. *qims* correspondingly gives the *qims* of the new indices, and *dirs* gives the new directions.

An example clarifies: Suppose *self* has *shape* `[dim1, dim2, dim3, dim4]`, *qhape* `[qim1, qim2, qim3, qim4]`, and *dirs* `[d1,d2,d3,d4]`. Suppose then that `indices = [1,3]`, `dims = [[dimA, dimB], [dimC, dimD]]`, `qims = [[qimA, qimB], [qimC, qimD]]` and `dirs = [[dA, dB] [dC, dD]]`. Then the resulting tensor will have *shape* `[dim1, dimA, dimB, dim3, dimC, dimD]`, *qhape* `[qim1, qimA, qimB, qim3, qimC, qimD]`, and *dirs* `[d1, dA, dB, d3, dC, dD]`. All this assuming that that *dims* and *qims* are such that joining *qimA* and *qimB* gives *qim2*, etc.

Instead of a list of indices a single index may be given. Correspondingly *dims*, *qims* and *dirs* should then have one level of depth less as well.

split_indices does not modify the original tensor, but returns a copy or a view.

sum()
Return the sum of all elements.

swapaxes(*i*, *j*)
Swap two indices, return a view.

to_ndarray()
Return a corresponding numpy array.

The order of the blocks in the result is such that along every index the blocks are organized according to rising *qnum*. Note that this means that the end result changes if the directions of some of the indices are flipped before calling *to_ndarray*. Thus if for example *trace* or *dot* is called on the resulting NumPy array, the result may be different than for the *AbelianTensor* if the contraction requires flipping directions. Similarly taking for example *traces* and *diags* along axes that were not compatible in the *AbelianTensor* is a perfectly valid thing to do for the *ndarray*, and gives different results.

All these concerns can be avoided by making sure that one only calls on the *ndarray* operations that would have been valid on the *AbelianTensor* without flipping any directions.

trace(*axis1=0*, *axis2=1*)
Take a trace over *axis1* and *axis2*.

This differs from the usual trace in the sense that it is more like connecting the two indices and contracting. This means that if the indices *axis1* and *axis2* don't have the same *dim* and *qim* the function will raise an error. If the *dirs* don't match (both are 1 or both are -1) then one of them is flipped and a warning is raised.

Note that the diagonal consists always of blocks with the same *qnum* on *axis1* and *axis2* (once *dirs* are opposite). This means that the trace of an invariant `charge != 0` tensor is always a zero-tensor.

transpose(*p=(1*, *0)*)
Transpose indices, return a view.

The optional argument *p* should be a permutation of all the indices. By default `p=(1, 0)`, which is the transpose of a matrix.

value()
For a scalar tensor, return the scalar.

view()
Return a view of this tensor.

A view is otherwise independent but identical to the original, but its *sects* points to the same numpy arrays as the sects of the original. In other words changing a whole block is ok, but modifying a block in place modifies the original as well.

CHAPTER 3

Tensor

class abeliantensors.tensor.**Tensor**

Bases: *abeliantensors.tensorcommon.TensorCommon*, numpy.ndarray

A wrapper class for NumPy arrays.

This class implements no new functionality beyond NumPy arrays, but simply provides them with the same interface that is used by the symmetry preserving tensor classes. *Tensors* always have `qhape == None`, `dirs == None` and `charge == 0`.

Note that *Tensor* is a subclass of both *TensorCommon* and *numpy.ndarray*, so many NumPy functions work directly on *Tensors*. It's preferable to use methods of the *Tensor* class instead though, because it allows to easily switching to a symmetric tensor class without modifying the code.

abs()
Return the element-wise absolute value.

all(**args*, ***kwargs*)
Return whether all elements are True.

See *numpy.ndarray.all* for details.

allclose(*other*, **args*, ***kwargs*)
Return whether self and other are nearly element-wise equal.

See *numpy.allclose* for details.

any(**args*, ***kwargs*)
Return whether any elements are True.

See *numpy.ndarray.any* for details.

average()
Return the element-wise average.

classmethod check_form_match(*tensor1=None*, *tensor2=None*, *qhape1=None*, *shape1=None*, *dirs1=None*, *qhape2=None*, *shape2=None*, *dirs2=None*, *qodulus=None*)
Check that the given two tensors have the same form in the sense that, i.e. that their indices have the same

dimensions. Instead of giving two tensors, two shapes can also be given.

compatible_indices(*other*, *i*, *j*)
Return True if index *i* of *self* and index *j* of *other* are of the same dimension.

conjugate()
Return the complex conjugate.

diag(***kwargs*)
Return the diagonal of a given matrix, or a diagonal matrix with the given values on the diagonal.

dot(*B*, *indices*)
Dot product of tensors.

See *numpy.tensordot* on how to use this, the interface is exactly the same, except that this one is a method, not a function. The original tensors are not modified.

exp()
Return the element-wise exponential.

expand_dims(*axis*, *direction=1*)
Add to *self* a new singleton index, at position *axis*.

classmethod eye(*dim*, *qim=None*, *qodulus=None*, **args*, ***kwargs*)
Return the identity matrix of the given dimension dim.

fill(*value*)
Fill the tensor with a scalar value.

flip_dir(*axis*)
A no-op, that returns a view.

The corresponding method of *AbelianTensor* flips the direction of an index, but directions are meaningless for *Tensors*.

classmethod from_ndarray(*a*, ***kwargs*)
Given an NumPy array, return the corresponding *Tensor* instance.

imag()
Return the imaginary part.

classmethod initialize_with(*numpy_func*, *shape*, **args*, *qhape=None*, *charge=None*, *invar=None*, *dirs=None*, ***kwargs*)
Use the given *numpy_func* to initialize a tensor of *shape*.

isscalar()
Return whether this *Tensor* is a scalar.

join_indices(**inds*, *return_transposed_shape_data=False*, ***kwargs*)
Join indices together in the spirit of reshape.

inds is either a iterable of indices, in which case they are joined, or a iterable of iterables of indices, in which case the indices listed in each element of *inds* will be joined.

Before any joining is done the indices are transposed so that for every batch of indices to be joined the first remains in place and the others are moved to be after in the order given. The order in which the batches are given does not matter.

If *return_transposed_shape_data* is True, then the shape of the tensor after transposing but before reshaping is returned as well, in addition to None and None, that take the place of *transposed_qhape* and *transposed_dirs* of *AbelianTensor*.

The method does not modify the original tensor.

log()
Return the element-wise natural logarithm.

matrix_dot(*B*)
Take the dot product of two tensors of order < 3 (i.e. vectors or matrices).

matrix_eig(*chis=None*, *eps=0*, *print_errors='deprecated'*, *hermitian=False*, *break_degenerate=False*, *degeneracy_eps=1e-06*, *sparse=False*, *trunc_err_func=None*)
Find eigenvalues and eigenvectors of a matrix.

The input must be a square matrix.

If *hermitian* is True the matrix is assumed to be hermitian.

Truncation works like for SVD, see the documentation there for more.

If *sparse* is True, a sparse eigenvalue decomposition, using power methods from *scipy.sparse.eigs* or *eigsh*, is used. This decomposition is done to find `max(chis)` eigenvalues, after which the decomposition may be truncated further if the truncation error so allows. Thus `max(chis)` should be much smaller than the full size of the matrix, if *sparse* is True.

The return values is `S, U, rel_err`, where *S* is a vector of eigenvalues and *U* is a matrix that has as its columns the eigenvectors. *rel_err* is the truncation error.

matrix_svd(*chis=None*, *eps=0*, *print_errors='deprecated'*, *break_degenerate=False*, *degeneracy_eps=1e-06*, *sparse=False*, *trunc_err_func=None*)
Singular value decompose a matrix.

The optional argument *chis* is a list of bond dimensions. The SVD is truncated to one of these dimensions *chi*, meaning that only *chi* largest singular values are kept. If *chis* is a single integer (either within a singleton list or just as a bare integer) this dimension is used. If `eps == 0`, the largest value in *chis* is used. Otherwise the smallest *chi* in *chis* is used, such that the relative error made in the truncation is smaller than *eps*. The truncation error is by default the Frobenius norm of the difference, but can be specified with the keyword agument *trunc_err_func*.

An exception to the above is made by degenerate singular values. By default truncation is never done so that some singular values are included while others of the same value are left out. If this is about to happen, *chi* is decreased so that none of the degenerate singular values are included. This default behavior can be changed with the keyword argument *break_degenerate*. The default threshold for when singular values are considered degenerate is 1e-6. This can be changed with the keyword argument *degeneracy_eps*.

If *sparse* is True, a sparse SVD, using power methods from *scipy.sparse.svds*, is used. This SVD is done to find `max(chis)` singular values, after which the decomposition may be truncated further if the truncation error so allows. Thus `max(chis)` should be much smaller than the full size of the matrix, if *sparse* is True.

The return value is''U, S, V, rel_err'', where *S* is a vector and *U* and *V* are unitary matrices. They are such that `U.diag(S).V == self`, where the equality is appromixate if there is truncation. *rel_err* is the truncation error.

multiply_diag(*diag_vect*, *axis*, **args*, ***kwargs*)
Multiply by a diagonal matrix on one axis.

The result of *multiply_diag* is the same as `self.dot(diag_vect.diag(), (axis, 0))` This operation is just done without constructing the full diagonal matrix.

real()
Return the real part.

sign()
Return the element-wise sign.

split_indices(*indices*, *dims*, *qims=None*, ***kwargs*)
Splits indices in the spirit of reshape.

indices is an iterable of indices to be split. *dims* is an iterable of iterables such that `dims[i]` is an iterable of lists of dimensions, each list giving the dimensions along a new index that will come out of splitting `indices[i]`.

An example clarifies: Suppose *self* has *shape* `[dim1, dim2, dim3, dim4]`. Suppose then that `indices = [1,3]`, `dims = [[dimA, dimB], [dimC, dimD]]`. Then the resulting tensor will have `shape = [dim1, dimA, dimB, dim3, dimC, dimD]`, assuming that that *dims* and are such that joining *dimA* and *dimB* gives *dim2*, etc.

Instead of a list of indices a single index may be given. Correspondingly *dims* should then have one level of depth less as well.

split_indices never modifies the original tensor.

sqrt()
Return the element-wise square root.

sum()
Return the element-wise sum.

to_ndarray(***kwargs*)
Return the corresponding NumPy array, as a copy.

trace(*axis1=0*, *axis2=1*)
Return the trace over indices *axis1* and *axis2*.

value()
For a scalar tensor, return the scalar. For a non-scalar one, raise a *ValueError*.

CHAPTER 4

TensorCommon

class abeliantensors.tensorcommon.**TensorCommon**

Bases: `object`

A base class for *Tensor* and *AbelianTensor*, that implements some higher level functions that are common to the two.

Useful also for type checking as in `isinstance(T, TensorCommon)`.

classmethod default_trunc_err_func(*S*, *chi*, *norm_sq=None*)

The default error function used when truncating decompositions: L_2 norm of the discarded singular or eigenvalues `S[chi:]`, divided by the L_2 norm of the whole spectrum *S*.

A keyword argument *norm_sq* gives the option of specifying the Frobneius norm manually, if for instance *S* isn't the full spectrum to start with.

dot(*other*, *indices*)

Dot product of tensors.

See *numpy.tensordot* on how to use this, the interface is exactly the same, except that this one is a method, not a function. The original tensors are not modified.

eig(*a*, *b*, **args*, *return_rel_err=False*, ***kwargs*)

Eigenvalue decompose the tensor.

Transpose indices *a* to be on one side of *self*, *b* on the other, and reshape *self* to a matrix. Then find the eigenvalues and eigenvectors of this matrix, and reshape the eigenvectors to have on the left side the indices that *self* had on its right side after transposing but before reshaping.

If the keyword argument *hermitian* is True then the matrix that is formed after the reshape is assumed to be hermitian.

Truncation works like with SVD.

If the keyword argument *sparse* is True, a sparse eigenvalue decomposition, using power methods from *scipy.sparse.eigs* or *eigsh*, is used. This decomposition is done to find `max(chis)` eigenvalues, after which the decomposition may be truncated further if the truncation error so allows. Thus `max(chis)` should be much smaller than the full size of the matrix, if *sparse* is True.

Output is `S, U, [rel_err]`, where *S* is a vector of eigenvalues and *U* is a tensor such that the last index enumerates the eigenvectors of *self* in the sense that if `u_i = U[...,i]` then `self.dot(u_i, (b, all_indices_of_u_i)) == S[i] * u_i`. *rel_err* is relative error in truncation, only returned if *return_rel_err* is True.

The above syntax is precisely correct only for *Tensors*. For *AbelianTensors* the idea is the same, but eigenvalues and vectors come with quantum numbers so the syntax is slightly different. See *AbelianTensor.matrix_eig* for more details about what precisely happens.

The original tensor is not modified by this method.

classmethod empty(**args*, ***kwargs*)
Initialize a tensor of given form with *np.empty*.

static flatten_dim(*dim*)
Given a *dim* for a single index that may be divided between sectors, return a flattened dim, that has just the total dimension of the index.

static flatten_shape(*shape*)
Given a *shape* that may have dimensions divided between sectors, return a flattened shape, that has just the total dimension of each index.

form_str()
Return a string that describes the form of the tensor: the *shape*, *qhape* and *dirs*.

from_matrix(*left_dims*, *right_dims*, *left_qims=None*, *right_qims=None*, *left_dirs=None*, *right_dirs=None*)
Reshape a matrix back into a tensor, given the form data for the tensor.

The counter part of *to_matrix*, *from_matrix* takes in a matrix and the *dims*, *qims* and *dirs* lists of the left and right indices that the resulting tensor should have. Mainly meant to be used so that one first calls *to_matrix*, takes note of the *transposed_shape_data* and uses that to reshape the matrix back to a tensor once one is done operating on the matrix.

norm()
Return the Frobenius norm of the tensor.

norm_sq()
Return the Frobenius norm squared of the tensor.

classmethod ones(**args*, ***kwargs*)
Initialize a tensor of given form with *np.ones*.

classmethod random(**args*, ***kwargs*)
Initialize a tensor of given form with np.random.random_sample.

split(*a*, *b*, **args*, *return_rel_err=False*, *return_sings=False*, *weight='both'*, ***kwargs*)
Split the tensor into two with an SVD.

This is like an SVD, but takes the square root of the singular values and multiplies both unitaries with it, so that the tensor is split into two parts. Values are returned as `US, [S], SV, [rel_err]`, where the ones in square brackets are only returned if the corresponding arguments, *return_rel_err* and *return_sings*, are True.

The distribution of `sqrt(S)` onto the two sides can be changed with the keyword argument *weight*. If `weight="left"` (correspondingly `"right"`) then *S* is multiplied into *U* (correspondingly *V*). By default `weight="both"`, in which the square root is evenly distributed.

svd(*a*, *b*, **args*, *return_rel_err=False*, ***kwargs*)
Singular value decompose a tensor.

Transpose indices *a* to be on one side of *self*, *b* on the other, and reshape *self* to a matrix. Then singular value decompose this matrix into `U, S, V`. Finally reshape the unitary matrices to tensors that have a

new index coming from the SVD, for *U* as the last index and for *V* as the first, and *U* to have indices a as its first indices and *V* to have indices *b* as its last indices.

The optional argument *chis* is a list of bond dimensions. The SVD is truncated to one of these dimensions *chi*, meaning that only *chi* largest singular values are kept. If *chis* is a single integer (either within a singleton list or just as a bare integer) this dimension is used. If `eps == 0`, the largest value in *chis* is used. Otherwise the smallest *chi* in *chis* is used, such that the relative error made in the truncation is smaller than *eps*. The truncation error is by default the Frobenius norm of the difference, but can be specified with the keyword agument *trunc_err_func*.

An exception to the above is made by degenerate singular values. By default truncation is never done so that some singular values are included while others of the same value are left out. If this is about to happen, *chi* is decreased so that none of the degenerate singular values are included. This default behavior can be changed with the keyword argument *break_degenerate*. The default threshold for when singular values are considered degenerate is 1e-6. This can be changed with the keyword argument *degeneracy_eps*.

If the keyword argument *sparse* is True, a sparse singular value decomposition, using power methods from *scipy.sparse.svds*, is used. This decomposition is done to find `max(chis)` singular values, after which the decomposition may be truncated further if the truncation error so allows. Thus `max(chis)` should be much smaller than the full size of the matrix, if *sparse* is True.

If *return_rel_err* is True then the relative truncation error is also returned.

The return value is `U, S, V, [rel_err]`. Here *S* is a vector of singular values and *U* and *V* are isometric tensors (unitary if the matrix that is SVDed is square and there is no truncation). `U . S. diag() . V == self`, up to truncation errors.

The original tensor is not modified by this method.

to_matrix(*left_inds*, *right_inds*, *dirs=None*, *return_transposed_shape_data=False*)
Reshape the tensor into a matrix.

The reshape is done by transposing *left_inds* to one side of *self* and *right_inds* to the other, and joining these indices so that the result is a matrix. On both sides, before reshaping, the indices are also transposed to the order given in *left*/*right_inds*. If one or both of *left*/right_inds is not provided the result is a vector or a scalar.

dirs are the directions of the new indices. By default it is *[1,-1]* for matrices and *[1]* (respectively *[-1]*) if only *left_inds* (respectively *right_inds*) is provided.

If *return_transposed_shape_data* is True then the *shape*, *qhape* and *dirs* of the tensor after all the transposing but before reshaping is returned as well.

classmethod zeros(**args*, ***kwargs*)
Initialize a tensor of given form with *np.zeros*.

Index

A

AbelianTensor (*class in abeliantensors.abeliantensor*), 5
abs() (*abeliantensors.tensor.Tensor method*), 11
all() (*abeliantensors.abeliantensor.AbelianTensor method*), 6
all() (*abeliantensors.tensor.Tensor method*), 11
allclose() (*abeliantensors.abeliantensor.AbelianTensor method*), 6
allclose() (*abeliantensors.tensor.Tensor method*), 11
any() (*abeliantensors.abeliantensor.AbelianTensor method*), 6
any() (*abeliantensors.tensor.Tensor method*), 11
astype() (*abeliantensors.abeliantensor.AbelianTensor method*), 6
average() (*abeliantensors.abeliantensor.AbelianTensor method*), 6
average() (*abeliantensors.tensor.Tensor method*), 11

C

check_consistency() (*abeliantensors.abeliantensor.AbelianTensor method*), 6
check_form_match() (*abeliantensors.abeliantensor.AbelianTensor class method*), 6
check_form_match() (*abeliantensors.tensor.Tensor class method*), 11
check_qhape_shape_match() (*abeliantensors.abeliantensor.AbelianTensor class method*), 6
check_qim_dim_match() (*abeliantensors.abeliantensor.AbelianTensor class method*), 6
compatible_indices() (*abeliantensors.abeliantensor.AbelianTensor method*), 6
compatible_indices() (*abeliantensors.tensor.Tensor method*), 12
conj() (*abeliantensors.abeliantensor.AbelianTensor method*), 6
conjugate() (*abeliantensors.abeliantensor.AbelianTensor method*), 7
conjugate() (*abeliantensors.tensor.Tensor method*), 12
copy() (*abeliantensors.abeliantensor.AbelianTensor method*), 7

D

default_trunc_err_func() (*abeliantensors.tensorcommon.TensorCommon class method*), 15
defblock() (*abeliantensors.abeliantensor.AbelianTensor method*), 7
diag() (*abeliantensors.abeliantensor.AbelianTensor method*), 7
diag() (*abeliantensors.tensor.Tensor method*), 12
dot() (*abeliantensors.tensor.Tensor method*), 12
dot() (*abeliantensors.tensorcommon.TensorCommon method*), 15

E

eig() (*abeliantensors.tensorcommon.TensorCommon method*), 15
empty() (*abeliantensors.tensorcommon.TensorCommon class method*), 16
empty_like() (*abeliantensors.abeliantensor.AbelianTensor method*), 7
exp() (*abeliantensors.tensor.Tensor method*), 12
expand_dims() (*abeliantensors.abeliantensor.AbelianTensor method*), 7

expand_dims() (*abeliantensors.tensor.Tensor method*), 12
eye() (*abeliantensors.abeliantensor.AbelianTensor class method*), 7
eye() (*abeliantensors.symmetrytensors.TensorZN class method*), 3
eye() (*abeliantensors.tensor.Tensor class method*), 12

F

fill() (*abeliantensors.abeliantensor.AbelianTensor method*), 7
fill() (*abeliantensors.tensor.Tensor method*), 12
flatten_dim() (*abeliantensors.tensorcommon.TensorCommon static method*), 16
flatten_shape() (*abeliantensors.tensorcommon.TensorCommon static method*), 16
flip_dir() (*abeliantensors.abeliantensor.AbelianTensor method*), 7
flip_dir() (*abeliantensors.tensor.Tensor method*), 12
form_str() (*abeliantensors.tensorcommon.TensorCommon method*), 16
from_matrix() (*abeliantensors.tensorcommon.TensorCommon method*), 16
from_ndarray() (*abeliantensors.abeliantensor.AbelianTensor class method*), 7
from_ndarray() (*abeliantensors.symmetrytensors.TensorZN class method*), 3
from_ndarray() (*abeliantensors.tensor.Tensor class method*), 12

I

imag() (*abeliantensors.abeliantensor.AbelianTensor method*), 7
imag() (*abeliantensors.tensor.Tensor method*), 12
initialize_with() (*abeliantensors.abeliantensor.AbelianTensor class method*), 7
initialize_with() (*abeliantensors.symmetrytensors.TensorZN class method*), 4
initialize_with() (*abeliantensors.tensor.Tensor class method*), 12
is_full() (*abeliantensors.abeliantensor.AbelianTensor method*), 8
is_valid_key() (*abeliantensors.abeliantensor.AbelianTensor method*), 8
isscalar() (*abeliantensors.abeliantensor.AbelianTensor method*), 8
isscalar() (*abeliantensors.tensor.Tensor method*), 12

J

join_indices() (*abeliantensors.abeliantensor.AbelianTensor method*), 8
join_indices() (*abeliantensors.tensor.Tensor method*), 12

L

log() (*abeliantensors.tensor.Tensor method*), 12

M

matrix_dot() (*abeliantensors.abeliantensor.AbelianTensor method*), 8
matrix_dot() (*abeliantensors.tensor.Tensor method*), 13
matrix_eig() (*abeliantensors.abeliantensor.AbelianTensor method*), 8
matrix_eig() (*abeliantensors.tensor.Tensor method*), 13
matrix_svd() (*abeliantensors.abeliantensor.AbelianTensor method*), 8
matrix_svd() (*abeliantensors.tensor.Tensor method*), 13
max() (*abeliantensors.abeliantensor.AbelianTensor method*), 9
min() (*abeliantensors.abeliantensor.AbelianTensor method*), 9
multiply_diag() (*abeliantensors.abeliantensor.AbelianTensor method*), 9
multiply_diag() (*abeliantensors.tensor.Tensor method*), 13

N

norm() (*abeliantensors.tensorcommon.TensorCommon method*), 16
norm_sq() (*abeliantensors.tensorcommon.TensorCommon method*), 16

O

ones() (*abeliantensors.tensorcommon.TensorCommon class method*), 16

R

random() (*abeliantensors.tensorcommon.TensorCommon class method*), 16
real() (*abeliantensors.abeliantensor.AbelianTensor method*), 9
real() (*abeliantensors.tensor.Tensor method*), 13

S

sign() (*abeliantensors.tensor.Tensor method*), 13
split() (*abeliantensors.tensorcommon.TensorCommon method*), 16
split_indices() (*abeliantensors.abeliantensor.AbelianTensor method*), 9
split_indices() (*abeliantensors.symmetrytensors.TensorZN method*), 4
split_indices() (*abeliantensors.tensor.Tensor method*), 13
sqrt() (*abeliantensors.tensor.Tensor method*), 14
sum() (*abeliantensors.abeliantensor.AbelianTensor method*), 10
sum() (*abeliantensors.tensor.Tensor method*), 14
svd() (*abeliantensors.tensorcommon.TensorCommon method*), 16
swapaxes() (*abeliantensors.abeliantensor.AbelianTensor method*), 10

T

Tensor (*class in abeliantensors.tensor*), 11
TensorCommon (*class in abeliantensors.tensorcommon*), 15
TensorU1 (*class in abeliantensors.symmetrytensors*), 3
TensorZ2 (*class in abeliantensors.symmetrytensors*), 3
TensorZ3 (*class in abeliantensors.symmetrytensors*), 3
TensorZN (*class in abeliantensors.symmetrytensors*), 3
to_matrix() (*abeliantensors.tensorcommon.TensorCommon method*), 17
to_ndarray() (*abeliantensors.abeliantensor.AbelianTensor method*), 10
to_ndarray() (*abeliantensors.tensor.Tensor method*), 14
trace() (*abeliantensors.abeliantensor.AbelianTensor method*), 10
trace() (*abeliantensors.tensor.Tensor method*), 14
transpose() (*abeliantensors.abeliantensor.AbelianTensor method*), 10

V

value() (*abeliantensors.abeliantensor.AbelianTensor method*), 10
value() (*abeliantensors.tensor.Tensor method*), 14
view() (*abeliantensors.abeliantensor.AbelianTensor method*), 10

Z

zeros() (*abeliantensors.tensorcommon.TensorCommon class method*), 17